Universitat Politècnica de Catalunya

Master:

Mechanical Engineering

Final Project

Development of Local Geometrical Planning for omni-directional robot

Francesco Chialà

Supervisor: Raúl Suárez Feijóo

2018/19

September 2019

To my family...
for everything.

Abstract

The goal of this work is the development of an obstacle avoidance algorithm for the mobile platform of the Mobile Anthropomorphic Dual-Arm Robot (MADAR), designed by the *Institute of Industrial and Control Engineering* (IOC) and *Mechanical Engineering Department* at the Universidad Politècnica de Catalunya (UPC).

The algorithm takes as inputs laser measurements from the front scanner that the robot is equipped, and the vector of the goal to reach. A *local geometrical planning* is adopted, it is reactive and applied continuously each time that the scanner samples (frequency = 15 Hz). A heuristic is used to choose the direction of motion, which tries to execute the shorter path that the robot needs to cover in order to reach the goal.
It is validated the implementation with experimentation on the real robot.

In conclusion, the algorithm can be consider a possible solution for obstacle avoidance problems on every omni-directional robots or also, its logic could be reused as a part of other kind of algorithms to improve them.

Contents

List of Figures

1. Introduction

In this first chapter will be explain the motivation, the objectives and the software used.

1.1 Motivation

Collision avoidance is a very actual problem considering the larger use of *autonomous mobile robot* in industrial plants and also the growing interest in the *autonomous driving*.
It is applied to safety move robots in a fully unknown environment. It is important to say that the problem of obstacle avoidance is different from the so called path planning. In fact, the latter involves the pre-computation of a free-collision path knowing the map of the environment the robot navigates in. On the contrary, the distinctive feature of an obstacle avoidance algorithm is to be implemented as a reactive control using sensors which give information about the surrounding environment. In the fields of mobile robotics path planning, obstacle detection and collision avoidance work in order to plan a safe path towards a defined goal position avoiding collisions along the way.

1.2 Objectives

The present work focuses on the obstacle avoidance problem. It is proposed a *new real time avoidance algorithm* based on *local planning*. A heuristic is develop in order to take a decision in every situation could happen. Since the algorithm have to be reactive it is thought to make its as simple as possible, minimizing the use of loops and as consequence the computation time.

1.3 Software

The software used in this work includes: Robotic Operating System (ROS), C++ and Matlab. We will explain a bit what is ROS and how it works and the specific use of Matlab done.

ROS: Robotic Operating System

The definition commonly recognised of ROS is: "The Robot Operating System (ROS) is a flexible framework for writing robot software. It is a *collection of tools, libraries, and conventions* that aim to simplify the task of creating complex and robust robot behaviour across a wide variety of robotic platforms."

Figure 1.1: *ROS icon.*

We will try to described just some of the simple elements with which ROS works. We know that the engineering approach to solve a problem is to split it in a series of small problems. Each small problem can be saw as a process which takes in input some data, elaborates them and send in output other data which are read from the next process/processes. ROS allows to manage the communication between these small processes that are called **nodes**. The inputs/outputs are called **messages** which are written on **topics**. The topics could be saw as boards in which some data (messages) are written. All the other **nodes** could read if it is necessary. A node can publish a message on topics and/or read a message from topics. In order to do this we need to declare inside the script of the node some **publisher** and **subscriber** entities. Moreover, for each of them we need to specify the topic on/from which it has to publish/subscribe.

There are also **services** that are a sort of function which can be call inside a node in order to do some operations.

In our project we are used also a visualization tool provided by ROS whose name is "RViz" in order to see on the computer the laser detections instantly during the motion of the robot. In Figure 1.3 it is showed, on the left, the Mobile Anthropomorphic Dual-Arm Robot (MADAR) in RViz tool; on the right, the points cloud of the obstacle detected and the base link frame. In fact, in Rviz it is possible to hidden the elements which you do not desire to look in the visualization. For example, on the right picture of Figure 1.3 all the elements are hidden except for the *points cloud* and the *base link*. On the left it is enable the visualization of the entire robot only. The building of the robot is done declaring all its link and joint in .xml file called: URDF file (Universal Robotic Description File).

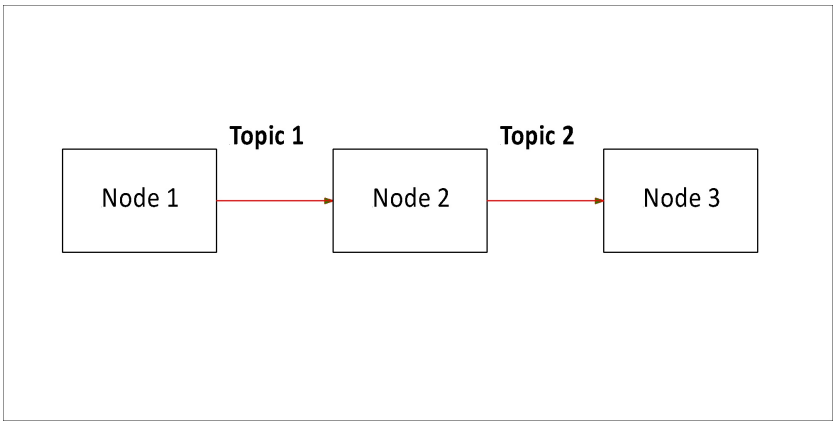

Figure 1.2: *ROS nodes and topics.*

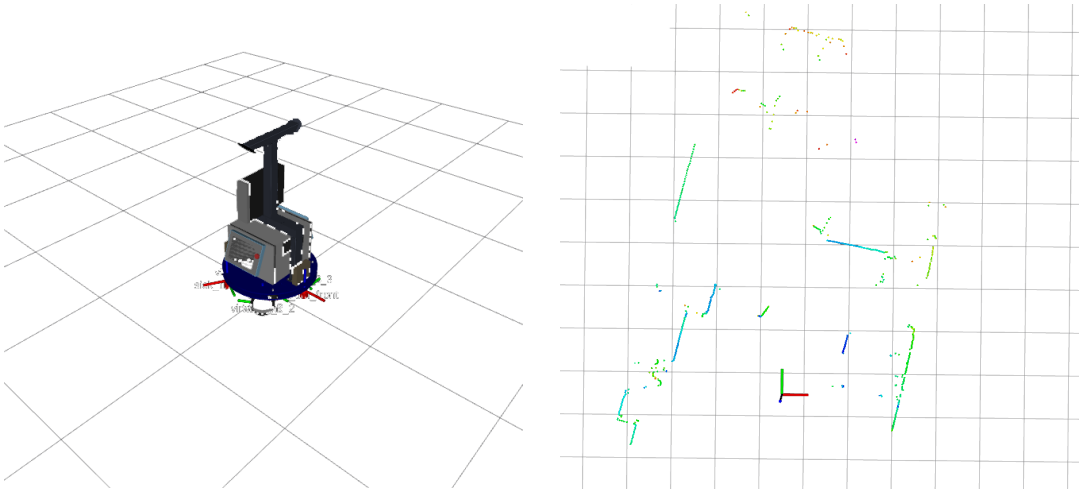

Figure 1.3: *Visualization of the robot.*

Matlab

A continuous check of the correctness of the algorithm is done during the implementation thanks to the visualization tool Rviz combined with the pictures made in Matlab after post-processing of the laser data (Figure 1.4).

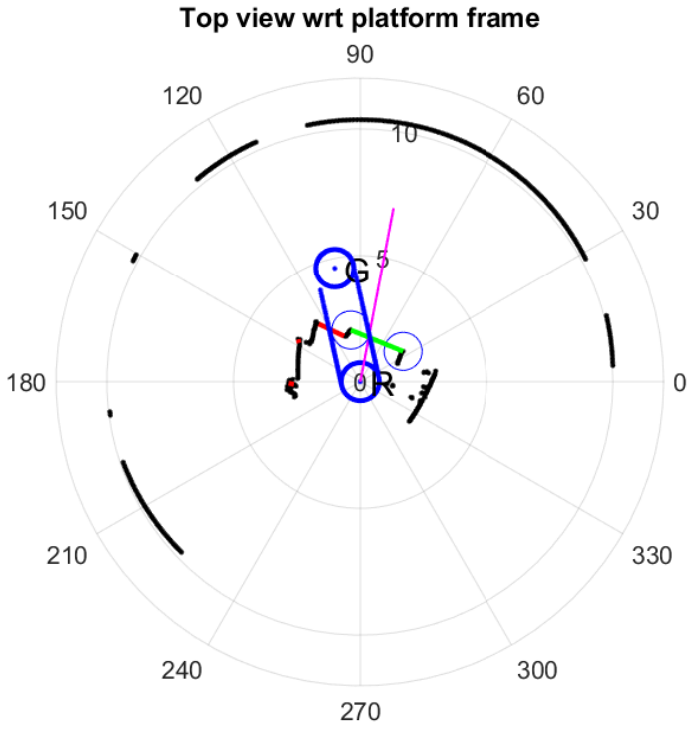

Figure 1.4: *Matlab picture made after post-processing of the laser's data.*

In Figure 1.4 there is a picture made after post-processing of the laser data. It is possible to match this picture with the left picture in Figure 1.3 finding here the gaps red and green showed in the Figure 1.4.

2. Hardware description

In this chapter will be described the hardware. At first, it will be given an overview of the prototype robot named MADAR (Mobile Anthropomorphic Dual-Arm Robot) and then it will be described in more details its *mobile platform* and its particular *omni-directional wheels*. Finally, considering the design of wheels, it will be presented the kinematics model of the platform computing the Jacobian matrix which allows to change the coordinates from the so called Joints Space to the Work Space (or Cartesian Space).

2.1 Mobile Manipulator

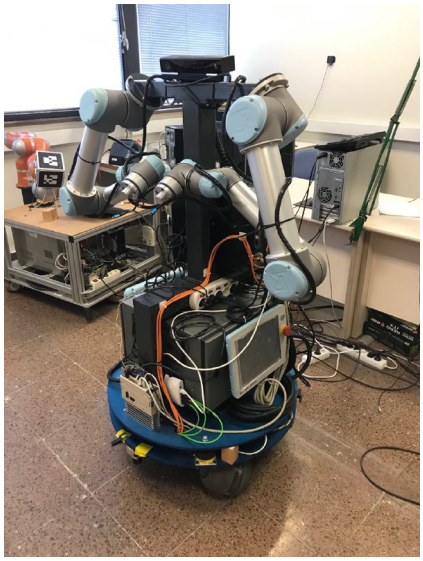

Figure 2.1: *MADAR: Mobile Anthropomorphic Dual-Arm Robot.*

Robotics is getting a larger success in a lot of fields and applications and the robot's companies are developing different kinds of robots for each of them. A typical case of study is the development of the so called **mobile**

manipulator. The term is used to refer to robot systems built from a manipulator arm mounted on a mobile platform. The advantage of this solution is to offer unlimited workspace to the manipulator. However, the manage of the systems is not easy because of the many degrees of freedom and the unstructured environment where the robot moves. This is the reason why a mobile manipulator needs a vision or laser system in order to perform its localization and its free collision movement. The mobile platform could have different kind of wheels depending on the desired performance and the expected conditions of the environment.

The robot presented in this work is a prototype named MADAR (from Mobile Anthropomorphic Dual-Arm Robot) in Figure 2.1. It has been designed thanks to the collaboration of Institute of Industrial and Control Engineering (IOC) and Mechanical Engineering Department at the Universidad Politècnica de Catalunya (UPC). It is composed by a dual-arm torso with a human-like structure assembled on an omnidirectional platform. The dual-arm system integrates two arms UR5 (6 degrees of freedom), each one equipped with Allegro Hand with four fingers and a total of 16 degrees of freedom (4 degrees of freedom for each finger).

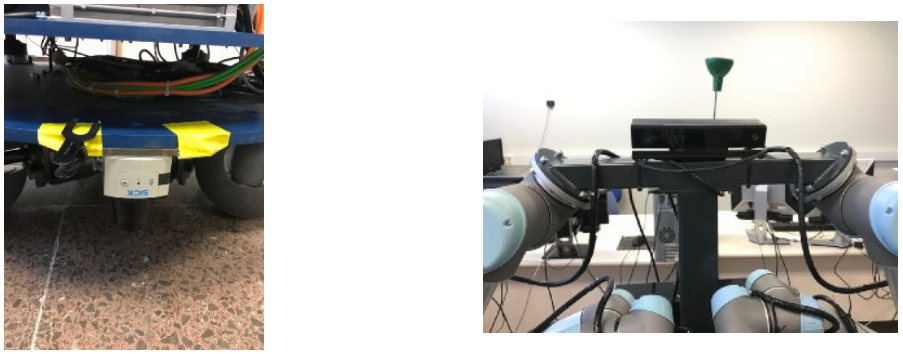

Figure 2.2: *Laser-ranges sensor on the left and RGB-D camera on the right.*

The mobile platform is circular with three special wheels which allow an omnidirectional displacements. The whole structure is equipped with laser-range sensors, a radio positioning system and an RGB-D camera essential to detect the environment (Figure 2.2).

2.2 Mobile platform

The presented work will focus on the mobile platform. Basically, there are two way to build a mobile robot. The first one is based on the use of conventional wheels (free rolling or driving wheels, steering-controlled wheels or caster wheels). The advantage of this approach is the simplicity in the wheel design but the disadvantage is that the movement is non-holonomic. This make more difficult their control and the maneuverability. On the contrary, the second approach is based on the use of non-conventional wheels or omnidirectional wheels. As consequence the control of the platform is more easy but the disadvange is that the design of the wheels is more difficult.
The prototype platform presented in this work follows the second approach and uses a special kind of omni-wheel which are described as follow.

2.3 Omni-directional wheels

2.3.1 Mecanum Wheels

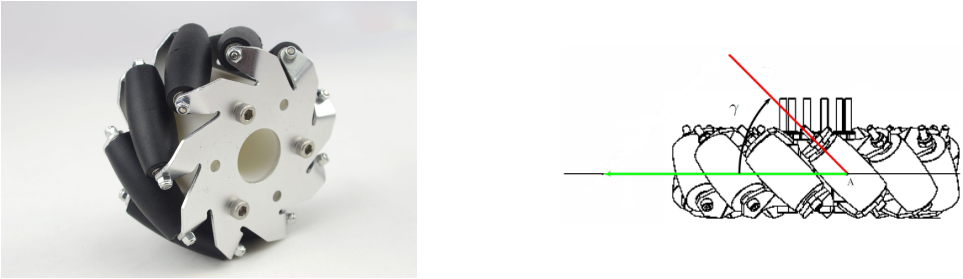

Figure 2.3: *Madar wheel prototype.*

The most common example of omnidirectional wheels are the *Mecanum Wheels* (Figure 2.3). The wheels used on MADAR are prototype but they are also omni-directional.

A Mecanum wheel could be consider a normal wheel surrounded on its circumference by a series of rollers with a particular orientation. Typically, in the *conventional* Mecanum wheels each roller has an axis of rotation at 45° to the plane of the wheel and at 45° to a line through the centre of the roller parallel to the axis of rotation of the wheel. This means that the angle γ in the Figure 2.3 is equal to 45°.

2.3.2 MADAR's wheels

The circular platform of the MADAR is equipped by three wheels placed in radial direction at 0.36 mm from the platform's center. The wheels developed are spherical omni-wheels composed of **spherical sectors** and rounded-shape **rollers with free-rolling movement** (Figure 2.4). Each wheel is commanded by a motor which provide torque around the **axis of the motor** (Figure 2.4). The motors are located with the shaft in the radial direction of the platform. All the other movements of the wheels are passive.

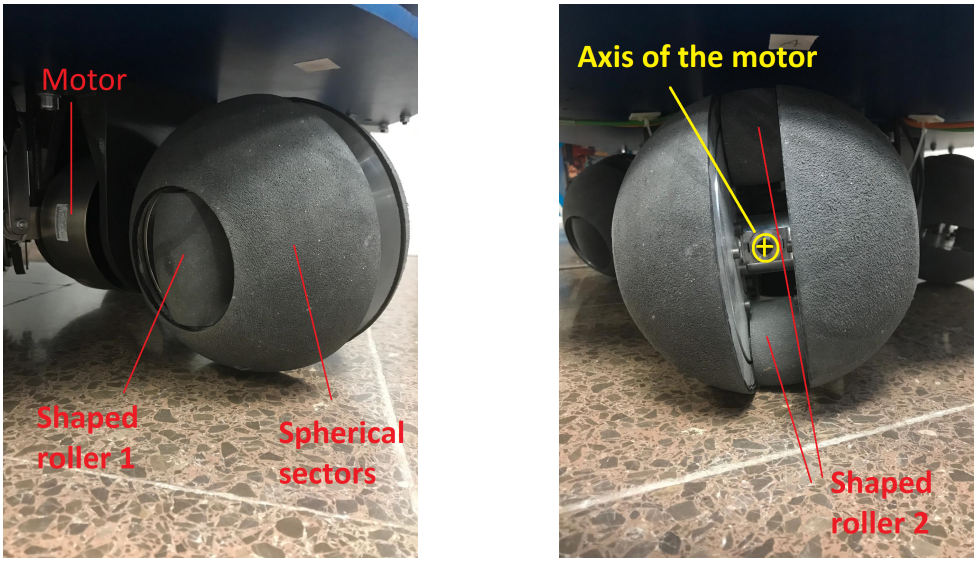

Figure 2.4: *Madar wheel prototype.*

2.4 Platform's Kinematics

The generalized position of the platform in a planar surface is express by three coordinates which give its position (x, y) and orientation (φ). We call $\mathbf{s}$, $\dot{\mathbf{s}}$, $\ddot{\mathbf{s}}$ the vector of the generalized position, velocity ed acceleration of the platform wrt global fixed frame (apex *0*):

$$\mathbf{s}^0 = \begin{bmatrix} x \\ y \\ \varphi \end{bmatrix}, \dot{\mathbf{s}}^0 = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\varphi} \end{bmatrix}, \ddot{\mathbf{s}}^0 = \begin{bmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{\varphi} \end{bmatrix} \tag{2.1}$$

The forward velocity kinematics of the platform wrt its local reference frame is given by the following equation (the $\dot{\mathbf{s}}$ without apex means that the

vector is expressed wrt the reference frame of the modelled system that is the frame of the base link of the platform with the axes x and y in Figure 2.5:

$$\dot{\mathbf{s}} = J \cdot \dot{\mathbf{q}} \tag{2.2}$$

The Jacobian matrix J contains the geometric information of the system and allows to make a change of the coordinates from the so called workspace (WS) to the so called joint-space (JS). In the analysed system the joint-space is the vectorial space which contains the position, velocity and acceleration of the wheels commanded by the motors.

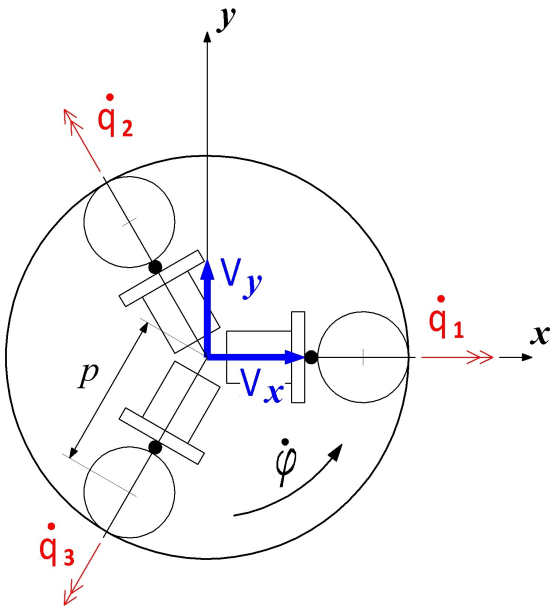

Figure 2.5: *Platform geometry.*

The kinematic model of the platform in Figure 2.5 produce the following matrix equation for the forward kinematics:

$$\dot{\mathbf{q}} = \begin{bmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \end{bmatrix} = 1/r \cdot \begin{bmatrix} 0 & -1 & -p \\ \sqrt{3}/2 & 1/2 & -p \\ -\sqrt{3}/2 & 1/2 & -p \end{bmatrix} \cdot \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\varphi} \end{bmatrix}$$

It is important to notice that in the system analysed the Jacobian matrix is constant and its determinant is always not null. As consequence, always exists the inverse of the matrix J. So, the inverse kinematics is given by:

$$\dot{\mathbf{q}} = J^{-1} \cdot \dot{\mathbf{s}} \tag{2.3}$$

3. Robot navigation

In this chapter it will be described the robot navigation and the steps you need to do in order to perform it. It is going to explain the mapping and localization processes in order to solve the so called SLAM (Simultaneous Localization and Mapping). Then it is going to talk about the Path Planning and the Obstacle Avoidance problems.

3.1 Overview

Nowadays a significant problem for such vehicles and mobile robots is the **obstacle detection and avoidance**. Actually, it should be the last step of a larger problem that is the **robot autonomous navigation**. In fact, in order to perform it you should solve:

1. **mapping** which provides the map of the environment where you want your robot to navigate at; it is usually made using two sources of information: the idiothetic (odometry) and the allothetic (vision) sources.

2. **localization** which allows you to localize the robot in the map;

3. **path planning** which provides the path and trajectory in the joint space in order to reach a goal in the work space;

4. **obstacle avoidance** which allows to avoid obstacles not found in the map;

3.2 Mapping

The first thing you need in order to perform robot autonomous navigation is a map of the environment where you want your robot to navigate at. The process which allows to get a map is called "Mapping Process".

ROS gmapping package

The mapping problem is already solved in ROS thanks to the *gmapping* package which perform SLAM. Basically, the package requires

- the transforms necessary to relate frames for laser, base, and odometry: odom → base link → laser
- laser scans to create the map

and provides:

- the current estimate of the robot's pose within the map frame that is the transform: map → odom

The gmapping package contains a node called **slam_gmapping**. It allows you to create a 2D map by using the data of the laser and of the robot's transforms and turns it into an **occupancy grid map** (OGM). So, what you need to do in order to create a map of the environment is:

1. configure the gmapping launch file;
2. launch the slam_gmapping node and move the robot around the environment;
3. the slam gmapping node get as an input lasers and transforms data building the map;
4. the generated map is published during the mapping process in the topic /map allowing its visualization in RViz.

The topic /map contains a massage of type **nav_msgs/OccupancyGrid** which describes the map as an **array of integers**. Particularly the integer 0 means that the space is completely free (grey color in RViz), 100 completely occupied (black color in RViz), and -1 completely unknown.

Once you created a map you need to save the map. In order to do this you it has to use the command:
rosrun map_server map_saver -f [insert map's name]
This command provides two file:

- .png which contains image of the map
- .yalm which contains the map's metadata such as the resolution, the file containing the image of the map, origin, etc.

Then you can provide the map for another node typing:
rosrun map_server map_server [map's name file].yaml
The map is published by means of two topics:

- /map_metadata topic which contains the map's metadata;
- /map topic which contains map's occupancy grid.

It is important to stand out that the map created in this way is a **static map** because it will not change anymore. This means that it can not contemplate obstacle placed after the mapping is done. So, if we have to plan a collision free motion we need to call an obstacle avoidance algorithm each time that we will find an obstacle not contemplated in the map. Moreover it is a **2D map** because the objects do not have height. So, for instance, it is not useful for drone navigation.

We can see picture below, which come from a simulation done in the tool of ROS named Gazebo. In the simulation is applied the *gmapping* package.

3.3 Localization

After you has built the map you need to localize the robot in that map. This process is known as "Localization Process". The Localization problem, like the mapping is solved in ROS in the gmapping package. We do not talk in details how the package works but we explain something about the info that the package needs in order to estimate the pose of the robot inside the map. The robot can provide two sources of information in order to get a map: the idiothetic and the allothetic sources:

- **allothetic information**: provided by lasers range finder, sonar or vision; the problem is that two different similar places could be perceived as the same (perceptual aliasing). For instance, it would be impossible to localize a robot in a building using only sensor measures because all the corridors may look the same;
- **idiothetic information**: corresponded to *odometry* which is a method to estimate the robot's pose wrt the starting location using data from motion sensors. The *odometry* is provided by the kinematics model of the robot. In order to simply explain what is odometry we give an example. We take the example of a 2d-problem of a wheel which turns without friction on a floor (Figure). The kinematics model in this case is very simple: $x = \theta \cdot r$. So, for example, when the wheel completes one turn the wheel's center of mass covers a length of $x = 2\pi r$ that is

the length of its circumference. In our case the kinematics model of the platform is what is explain in the chapter ?. However, allothetic information are subject to cumulative errors that grow quickly due by the not ideal condition in which the robot operates (for example friction in the motion).

Because of the limits of these information have, you need to combine the sources in order to compensate the errors for each other to some extent. This is what the package does to provide the pose of the robot.

3.4 Path planning

We will not discuss in details the Path Planning problem but we want only to remember some the approach usually used. In general there are two kind of planning:

1. **Joints Space planning**
   it consists to determine directly the motion laws $\mathbf{q(t)}$, $\dot{\mathbf{q}}\mathbf{(t)}$, $\ddot{\mathbf{q}}\mathbf{(t)}$

2. **Work Space planning**
   it consists to determine previously $\mathbf{s(t)}$, $\dot{\mathbf{s}}\mathbf{(t)}$, $\ddot{\mathbf{s}}\mathbf{(t)}$ and then with the inverse kinematics $\mathbf{q(t)}$, $\dot{\mathbf{q}}\mathbf{(t)}$, $\ddot{\mathbf{q}}\mathbf{(t)}$

In the mobile robots should be convenient to follow the Work Space planning. Then we can choose to plan with to kind of techniques:

1. **point to point**
   given $\mathbf{s}(t_{in})$, $\mathbf{s}(t_{fin})$ it is possible to plan a free-collision trajectory $\mathbf{s}(t)$ knowing that border condition $\dot{\mathbf{s}}(t_{in}) = \dot{\mathbf{s}}(t_{fin}) = 0$

2. **path motion**
   this kind of planning is used if it is necessary to go through some specific points given by $\mathbf{s}_1$, ..., $\mathbf{s}_i$, ..., $\mathbf{s}_n$. The result is to plan a piecewise trajectory $\mathbf{s}(t)$ between couple of points trying to choose borders conditions which make the trajectory function the as smooth as possible[1].

[1]The smoothness of the function is important to avoid jerk in the joints space.

3.5 Obstacle avoidance

Once an object is detected, the mobile robots must avoid it. Basically, there are two approaches to perform obstacle avoidance:

- **global approach**: the scheme is to inform the path planner of the obstacle and then let it re-plan the path.
  - PROS: the use of existing software for maneuvering the robot;
  - CONS: replanning is time consuming;
- **local approach**: the robots determines how to avoid the obstacle without help from the path planner.
  - PROS: faster response without requiring the computing resource of the path planner;
  - CONS: during local avoidance the vehicle ignores the global map of known obstacles and does not know to turn control back to the path planner if mission efficiency is adversely affected.

Of course, the best is to smart combine both the methods in order to catch the pros and discard the cons and efficiently complete required tasks. However, it this report it is proposed a logic that follow the local approach.

4. Algorithm explanation

In this chapter it is explain the algorithm developed. The proposed algorithm is reactive and it is applied continually each time that the lasers samples. The mobile robot can be consider holonomic thanks to three omnidirectional wheels, so it can move in each direction without constraints.
It will be showed a scheme (Figure 4.1) useful to see the nodes created in ROS and their inputs/outputs. Then the algorithm itself is showed in a flow-chart form (Figure 4.2) and the operations done by each box will be explained.

4.1 ROS nodes

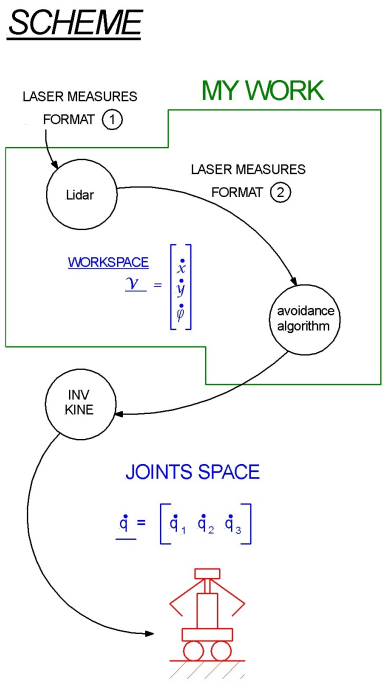

Figure 4.1: *Scheme for input output in ROS.*

The work done in this thesis regards what is surrounded by the green line in Figure 4.1. So two nodes are created:

- **lidar node**
  It has the role to change the format (format 1 $\rightarrow$ format 2) of the massage read by the topic of the front laser scan and filter a little bit the measures of the laser.

- **avoidance_algorithm node**
  Here it is developed the logic of the algorithm itself. Getting the laser measures in the format 2 provided by the *lidar node* it is produced the velocity in the workspace $\vec{\mathcal{V}} = [\dot{x},\ \dot{y},\ \dot{\varphi}]^T$.

After that, the generalized velocity $\vec{\mathcal{V}}$ (in the workspace) is sent to a node which applied the inverse kinematics and computes the torques in order to produce $\dot{q}$ (in the joints space) to give to the three motor in order to produce the desired $\vec{\mathcal{V}}$.

4.2 Flow chart

The flow chart of the algorithm is showed in Figure 4.2. It is important to notice that the only inputs of the algorithm are:

1. *instant laser measurements*
   They give in some sense a photo of the environment around the robot.

2. *goal vector wrt world frame* $\vec{G_{world}}$
   The frame world is a fixed frame automatically set in the position in which the motors are turned on.

On the other hand, the output is just one: *generalized velocity in the workspace* $\vec{\mathcal{V}}$. Finally, applying the inverse kinematics, the *velocity in the joints space* $\dot{\boldsymbol{q}}$ will be sent to the motor.

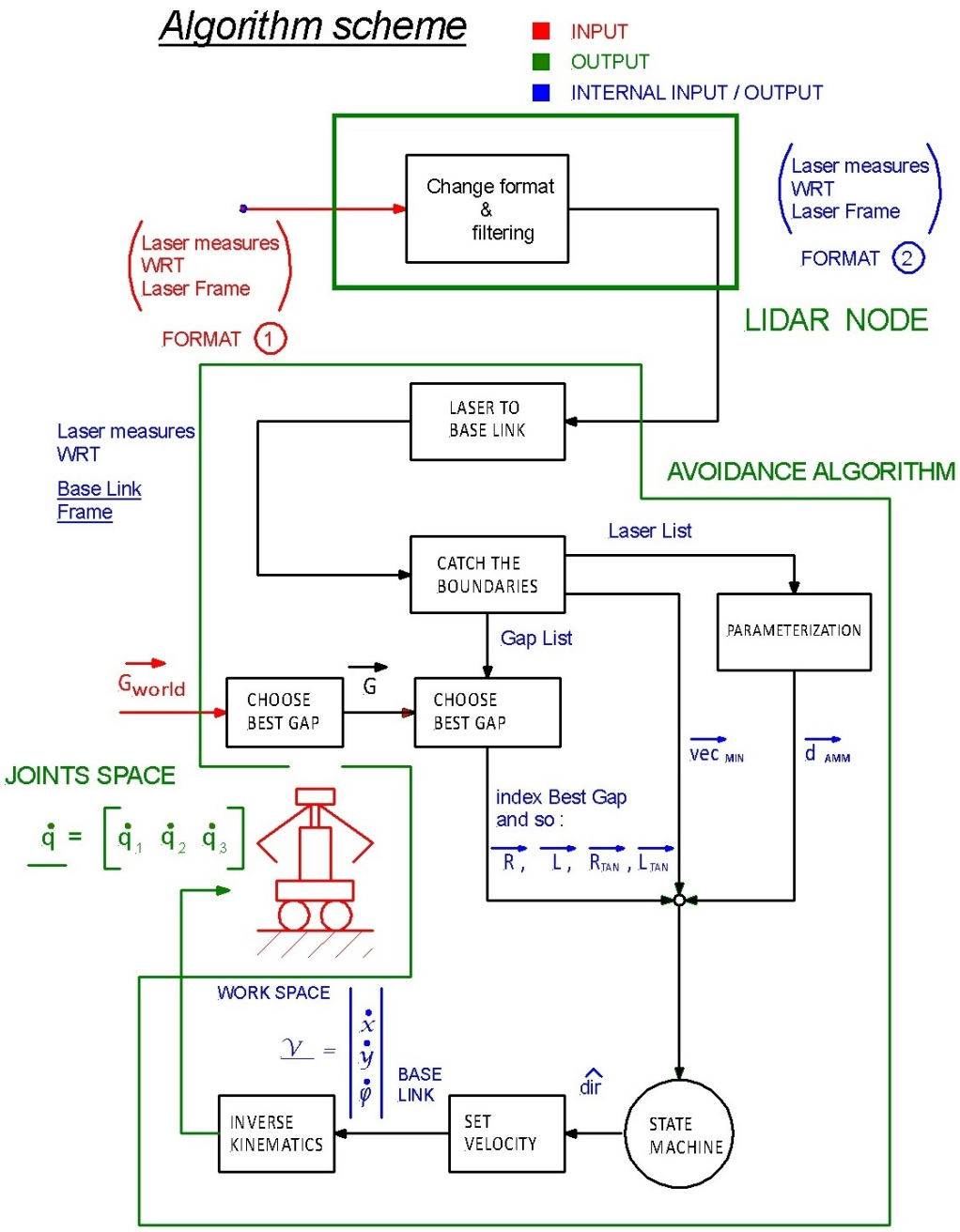

Figure 4.2: *Flow chart of the algorithm.*

4.3 Change format and filtering

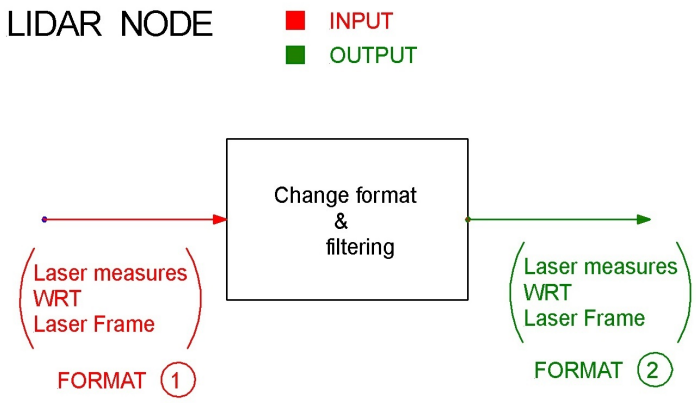

Figure 4.3: *Box change format and little filtering.*

The box *Change format and filtering* is the first box in the flow-chart in Figure 4.2 and, at the same time, the only box of the *lidar node*. In Figure 4.3 is showed the box with its input and output. In the following sections it will be discuss separately what is done for how is *changed the format* and how is applied the *filtering*.

Change format

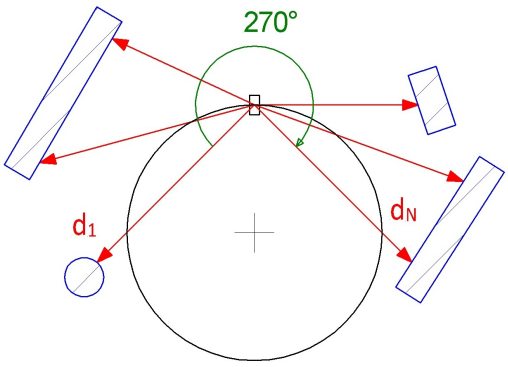

Figure 4.4: *Angle range of the laser.*

The format of the massage in input is an array of numbers (float):

$$inputData = [d_1, .., d_i, .., d_N]_{LASER} \tag{4.1}$$

It is necessary to convert it in an array of vectors:

$$vectorDataList = [\vec{d_1}, .., \vec{d_i}, .., \vec{d_N}]_{LASER} \tag{4.2}$$

One of the feature of the laser is to cover a range of 270°. So knowing this it is possible to compute the *resolution* of the laser as:

$$resolution = 270/size(inputData) \tag{4.3}$$

Knowing the resolution it is possible to express the distance in a vector form wrt the laser frame with the following equation:

$$\vec{d_i} = \begin{bmatrix} d_{i,x} \\ d_{i,y} \end{bmatrix} = \begin{bmatrix} d_i \cdot sin(deg2rad \cdot \alpha_i) \\ -d_i \cdot cos(deg2rad \cdot \alpha_i) \end{bmatrix} \tag{4.4}$$

Where d is the distance measured by the laser and α_i is the angle corresponding which is incremented (with a step equal to the resolution) in a loop and cover all the range of the laser.

Jump and *gap* definition

In order to understand the filtering process it is important to give some definition which are used to create the gaps. At first we define *jump* as follow:

$$jump = |\vec{d_i}| - |\vec{d_{i-1}}| \tag{4.5}$$

A jump is useful to record a boundary's presence if its module is larger than a threshold Δ:

$$|jump| > \Delta \tag{4.6}$$

We can define two type of jumps (Figure 4.5):

- **left jump** if $jump < -\Delta$, it provides a left boundary vector $\vec{L}$;
- **right jump** if $jump > \Delta$, it provides a right boundary vector $\vec{R}$;

So, we can define the condition to create a gap: a gap is created every time that a *left jump* is followed by a *right jump*. So, we can define the gap as a vector (Figure 4.18):

$$\vec{gap} = \vec{L} - \vec{R} \tag{4.7}$$

Moreover, it is important to say that if the algorithm detects two consecutive *right* or *left jumps* no gap is considered because we can not know a priori if the passage is close or not. This concept is clear looking Figure 4.7.

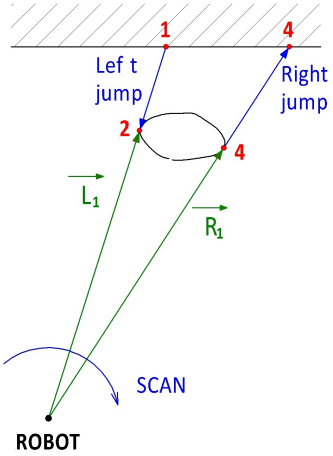

Figure 4.5: *Jump definition.*

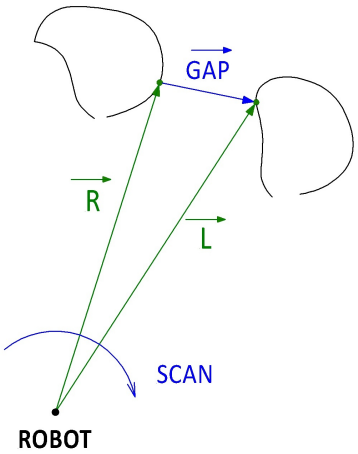

Figure 4.6: *Gap definition.*

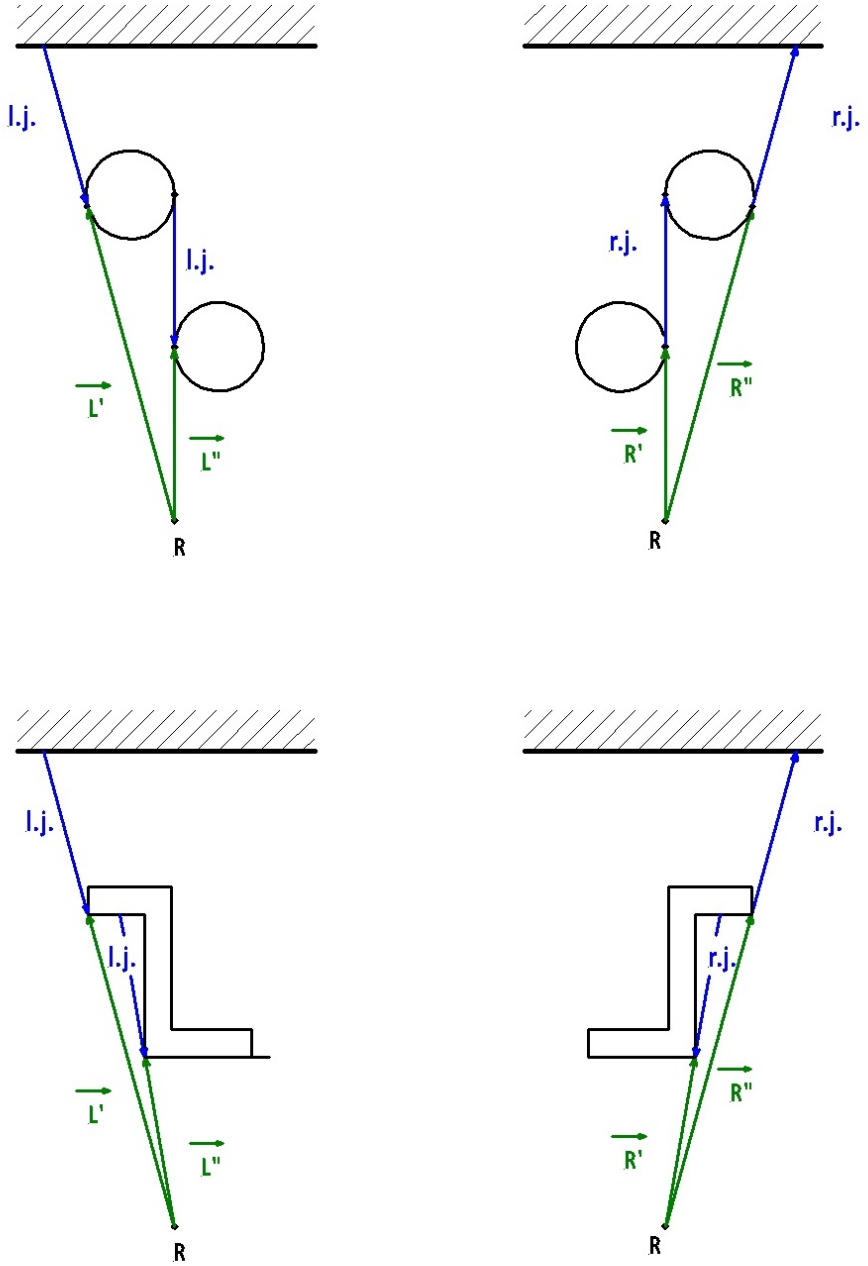

Figure 4.7: *No gap built.*

Filtering

Inside the lidar some filters are applied:

- filter 1: filter for short distances.
  It is necessary to do it because sometimes the laser detect some fake points very close to itself (Figure 4.8). We need to remove these points setting a distance parameter. Under that distance all the measure are discarded.

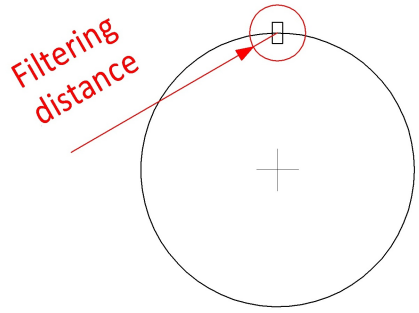

Figure 4.8: *Filter for short distances.*

- filter 2: filter for long distances.
  It is useful to create "artificially" the gap. In order to understand this point, look the following pictures remembering that, a gap is created catching a *right jump* followed by a *left jump*. Looking the examples in Figure 4.9 and Figure 4.10 it is possible to understand why this second filter is useful.

In Figure 4.9 and Figure 4.10 it is showed an example situation in which there is a room with only one obstacle inside. With R is indicated the robot, with G the goal point to reach. In the pictures the jumps are evidence with the color blue, $\vec{L}$ and $\vec{R}$ the vectors which give the position of the obstacle's boundaries wrt the robot.

In the example in Figure 4.9 the filter n.2 is not applied. The max length detectable by the laser is 10m. As consequence the shape detected by it is evidence in green in which the algorithm can catch the *left boundary followed by the right boundary* of the obstacle. As result no gaps can be built (remember the rules to build gaps)

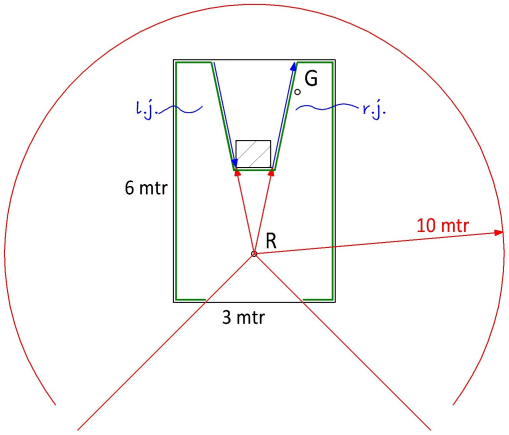

Figure 4.9: *Without filter 2.*

On the contrary, in the example in Figure 4.10 the filter n.2 is applied. The filter works with the following relation:

$$\begin{aligned} if(distance\_detected > distance\_looked) \\ distance = large\_value \end{aligned} \tag{4.8}$$

The parameter $distance\_looked$ should be setting considering the dimension of the space where the robot moves and the density of the obstacle. In the experiment done the parameter $distance\_looked = 3m$ and $large\_value = 10m$. The large value is used to create a jump artificially. In fact each time that the Equation 4.8 happens, a jump (of type *left* or *right*) is saw from the algorithm. Comparing Figure 4.9 and Figure 4.10 we can see that in the second example are artificially created the *r.j.* and the *l.j.* on the side evidenced in yellow. In other words, we can observe that using the filter n.2 (Figure 4.10) it is possible to create useful gaps for the robot (see GAP1 and GAP2).

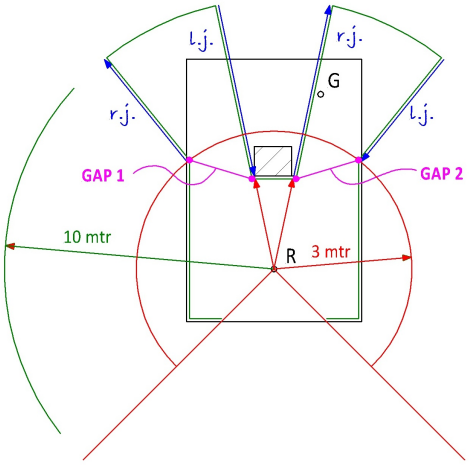

Figure 4.10: *Filter 2 applied.*

4.4 Laser frame to base_link frame

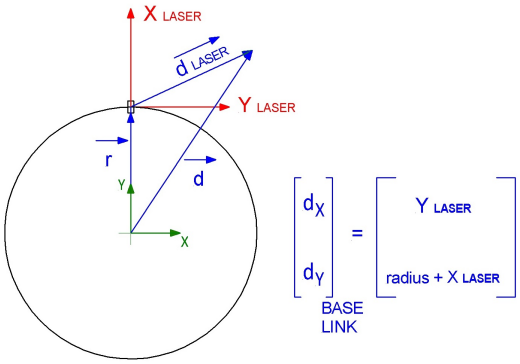

Figure 4.11: *Transformation of from laser frame to base_link frame.*

This box is very simple. It transforms the all the vectors of the *vectorData* form the *laser frame* to the *base_link frame* located on the centre of the platform with the y-axis pointing in the front of the robot. The output of the box is an array of vector wrt the *base_link frame*. Looking Figure 4.11 it

is easy to write the transformation equation:

$$\begin{bmatrix} d_x \\ d_y \end{bmatrix}_{base\_link} \quad \begin{bmatrix} y_{LASER} \\ radius + x_{LASER} \end{bmatrix} \tag{4.9}$$

Where d_x and d_y are the component of the distance detected by the laser wrt the frame *base_link*, *radius* is the radius of the platform (0.36m), x_{LASER} and y_{LASER} are the component of a generic distance detected by the laser wrt the *laser frame*.

4.5 Catch the boundaries

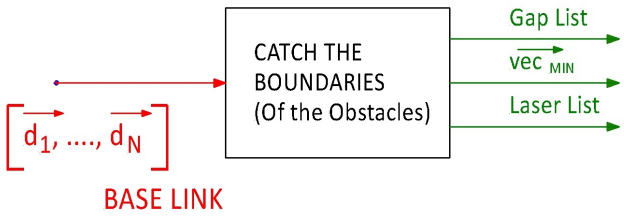

Figure 4.12: *Catch the boundaries box.*

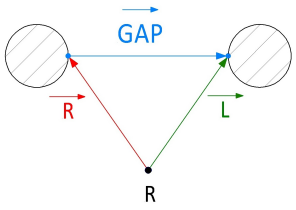

Figure 4.13: *Gap's definition:* $\vec{gap} = \vec{L} - \vec{R}$.

In this box is computed the logic in order to catch the points we need to create the gap. They should be real boundaries of obstacle or artificial boundaries (if they are produced by the filter n.2 inside the lidar node). The logic is based on the jump in magnitude that we can find inside the array of

vectors *vectorDataList* after its change of reference frame (form laser frame to base link frame). The output of this box are the following three variables:

$$gapList = \begin{bmatrix} \vec{R_1} & \vec{L_1} \\ .. & .. \\ \vec{R_i} & \vec{L_i} \\ .. & .. \\ \vec{R_N} & \vec{L_N} \end{bmatrix} \tag{4.10}$$

$$v_{\vec{MIN}} \tag{4.11}$$

$$laserList = \begin{bmatrix} d_{AMM,1} & \alpha_1 \\ .. & .. \\ d_{AMM,i} & \alpha_i \\ .. & .. \\ d_{AMM,N} & \alpha_N \end{bmatrix} \tag{4.12}$$

The meaning of the variables is the following:

- gapList: it contains in the lines two vectors which define respectively the right boundary of an obstacle $\vec{R}$ and a left boundary of an obstacle $\vec{L}$. In other words they define the gap detected between two obstacle according to the gap's definition (Figure 4.13);
- $\vec{vec_{MIN}}$: it is the vector with the smallest magnitude between all the laser measures;
- laserList: it contains on the lines info of magnitude and angle of the laser measures which have an angle that is $\alpha_1 <= \alpha_i <= \alpha_N$. This is useful to check if the robot can go directly to the goal. In order to do this it is done a parameterization of the border of the shape covered by the robot if it moved directly to the goal. We remind this discussion to one of the next sections.

4.6 World to base_link

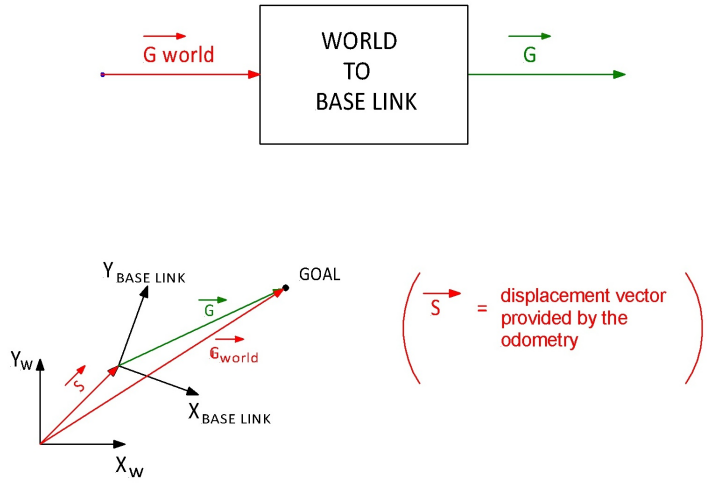

Figure 4.14: *Transformation from world_frame to base_link.*

This box is very simple because it does only the transformation from the world_frame to the base_link frame. The transformation is done taking the position of the two frames using the Ros function *listener*. In any case the transformation is simple and is given by the following vector equation:

$$\vec{G} = \vec{G_{world}} - \vec{s} \tag{4.13}$$

The vector $\vec{G_{world}}$, which is the input of all the algorithm, is the position of the goal wrt the *world frame*. The *world frame* is set in the position where the robot is when its motors are switched on. The vector $\vec{s}$ is the position of the robot wrt *world frame* estimated with the odometry. It is an estimation of the displacement of the robot using the encoder sensor of the motors. The vector $\vec{G}$ is the position of the goal wrt *base_link frame* and is the output of the box discussed in this section.

4.7 Choose Best gap

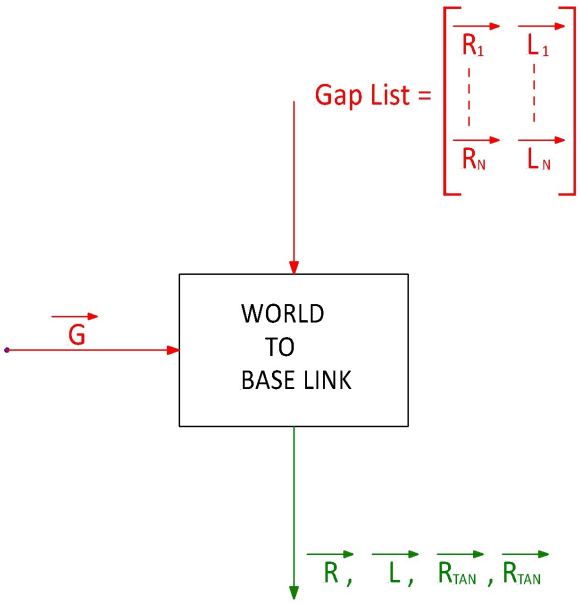

Figure 4.15: *Choose best gap box.*

The box provides the best gap that is: the gap which allows to cover the smallest distance to reach the goal according to the heuristic logic.

Remembering that the variable *gapList* contains line for line the information of the gaps found in the environment (in term of right $\vec{R}$ and left $\vec{L}$ boundaries vector), the choice of the best gap is no more then a index of the variable *Gap List*.

The logic in order to find the best gap is explained in Figure 4.16. In other words for each gap are computed the following quantities:

$$path_1 = |\vec{R}| + |\vec{G} - \vec{R}| \tag{4.14}$$

$$path_1 = |\vec{R}| + |\vec{G} - \vec{R}| \tag{4.15}$$

$$path_2 = |\vec{R} + \frac{\vec{gap}}{2}| + |\vec{G} - (\vec{R} + \frac{\vec{gap}}{2})|; \tag{4.16}$$

$$path_3 = |\vec{L}| + |\vec{G} - \vec{L}|. \tag{4.17}$$

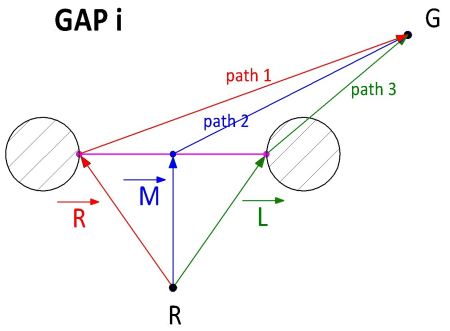

Figure 4.16: *Choice of the best gap.*

Where:

$$g\vec{a}p = \vec{L} - \vec{R}. \tag{4.18}$$

In order to choose the best gap is computing for each gap the quantity:

$$path = min(path_1, path_2, path_3). \tag{4.19}$$

It is decided to compute the smallest quantity between $path_1, path_2, path_3$ because in this way it is possible to give a good estimation of the shortest path to follow for a given gap regardless of the position of the goal wrt the robot (on the left, on the right, in the middle of the gap) like showed in the Figure 4.16. Then it is chosen the smallest *path* between all the minimum paths computed for each gap.

Moreover it is useful to compute the vectors $\vec{R_{tan}}$ $\vec{L_{tan}}$ tangent to a safety circles centered in the boundaries of the obstacles (Figure 4.17).

These vectors are computed solving the triangle evidence in Figure 4.17. Particularly, it is creating a function f_1 which allows to compute the rotational matrix in order to rotate a generic vector in the space given in input: *unit vector* around you want to rotate (for our problem the vector $\hat{k}$) and the angle of the rotation (α_{Dx} or α_{Sx}). Of course the angle has to follow the right hand convention around the unit vector.

$$\alpha_R = atan2\left(\frac{d_{safe}}{|\vec{R}|}\right); \tag{4.20}$$

$$\alpha_L = atan2\left(\frac{d_{safe}}{|\vec{L}|}\right); \tag{4.21}$$

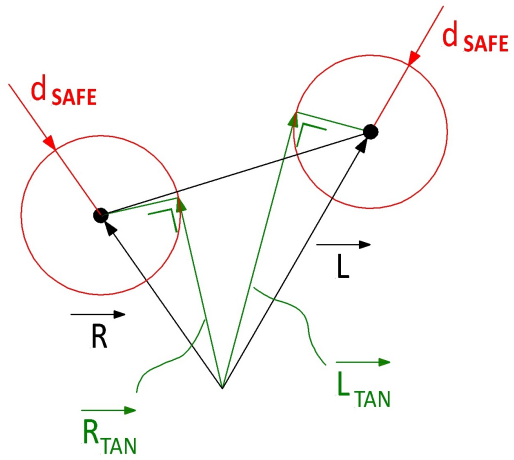

Figure 4.17: *Tangent vectors to the safe circle.*

Then with a second function f_2 which recieves as input the *vector which you want to rotate* and the *rotational matrix* it is possible to compute the vectors R_{tan} and L_{tan}. The succession of the operations are:

$$\begin{aligned} Rot_R &= f_1(\alpha_R, \hat{k}) \\ Rot_L &= f_1(\alpha_L, \hat{k}) \end{aligned} \tag{4.22}$$

$$\begin{aligned} \vec{R_{tan}} &= f_2(Rot_R, \vec{R}) \\ \vec{L_{tan}} &= f_2(Rot_L, \vec{L}) \end{aligned} \tag{4.23}$$

4.8 Parameterization

Figure 4.18: *Parameterization box.*

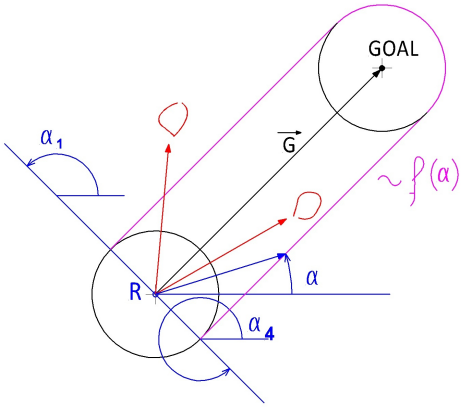

Figure 4.19: *Parameterization to compute the distance of the shape that the robot would cover if it moved directly to the goal.*

The parameterization is a piecewise function of the variable α. It allows to create an array of distances and angles (the angle is referred to the horizontal direction evidenced in black in the Figure 4.19) which are representative of the shape that the robot would cover if it moved directly to the goal. The distance computed are called *distance admissible* because, like we are going to explain in section "STATE MACHINE", the robot could move directly to the goal IF:

$$\forall \ \alpha \ \mid \ \alpha_1 < \alpha < \alpha_4 \ \exists \ d_{adm} < d_{measure} \tag{4.24}$$

4.9 State machine

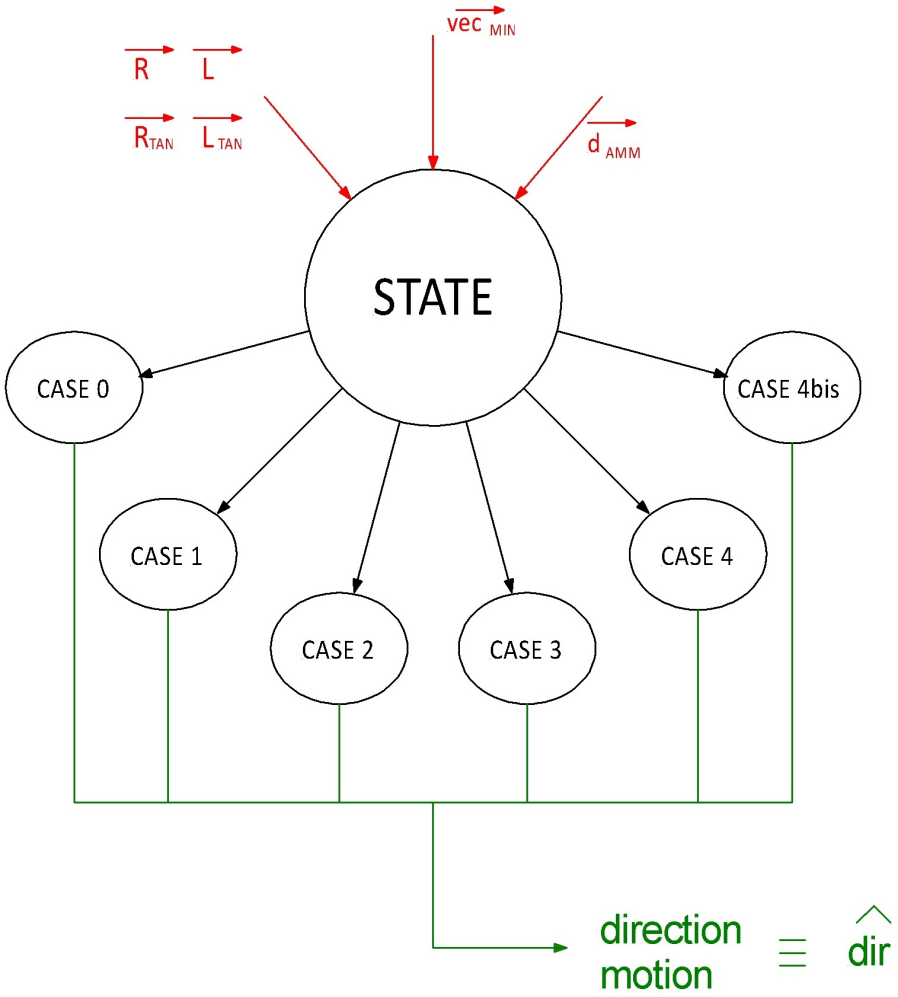

Figure 4.20: *State machine scheme.*

It is possible to model the system like a *state machine* with the graph in Figure 4.20. In order to enter in one of the cases some conditions have to be verify.

Case 0: robot close to the obstacle

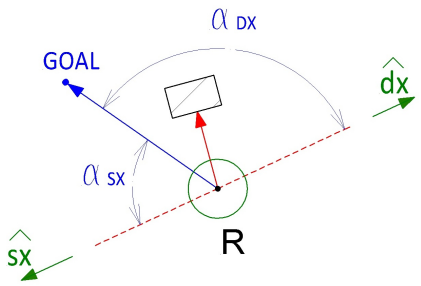

Figure 4.21: *Example case 0 state.*

The case 0 happens when the robot is very close to an obstacle. It means that there is at least one laser measure with a length detected under the safety distance d_{SAFE}. In this situation it is decided to move the robot in the direction orthogonal to the laser measure with the smallest magnitude $\vec{vec_{MIN}}$ In other words, in order to enter the condition to the case 0 is given by the following condition:

$$|\vec{vec_{MIN}}| <= d_{SAFE}. \tag{4.25}$$

Nevertheless, there are two direction orthogonal to $\vec{vec_{MIN}}$ which are identified with the unit vectors $\hat{sx}$ and $\hat{dx}$ (Figure 4.21).

The choice is done considering the angle between the unit vectors and the vector $\vec{G}$. These angles are computed with the following equations:

$$\begin{aligned} \alpha_{sx} &= |acos\left(\frac{\vec{G} \cdot \hat{sx}}{|\vec{G}|}\right)| \\ \alpha_{dx} &= |acos\left(\frac{\vec{G} \cdot \hat{dx}}{|\vec{G}|}\right)| \end{aligned} \tag{4.26}$$

and choosing the direction which has the smallest angle:

$$\begin{aligned} IF \ \ (\alpha_{dx} < \alpha_{sx}) \ \ &\rightarrow \ \ \hat{dx} \ \ choosed \\ ELSE \ \ &\rightarrow \ \ \hat{sx} \ \ choosed \end{aligned} \tag{4.27}$$

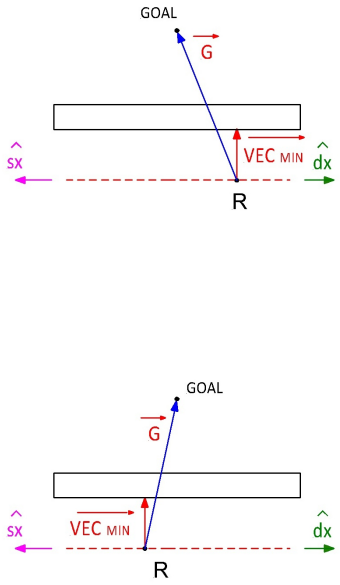

Figure 4.22: *Algorithm failure.*

For example in Figure 4.21 will be chosen the direction $\hat{sx}$ because $\alpha_{sx} < \alpha_{dx}$.

However there is a situation in which the algorithm could fail. It happens when the robot is very close to an obstacle so long. Even if the robot will move in an orthogonal direction the so much long obstacle does not allow the robot to detect a new gap. As consequence the robot will start to move in loop, towards right and left without avoid the obstacle (Figure 4.22). It is important to say that this situation could happen only if the robot is suddenly located near the so much long obstacle (for example it start to move very close to the obstacle or the obstacle suddenly appears during the robot movement). In fact, as we will explain in the next subsections, if the obstacle is far enough from the robot, it is able to move directly towards the gaps detected avoiding the obstacle without enter in *case 0* (Figure 4.23)

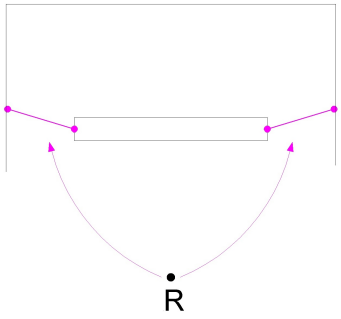

Figure 4.23: *Obstacle so much long but far enough from the robot.*

Case 1: robot can go directly to the goal

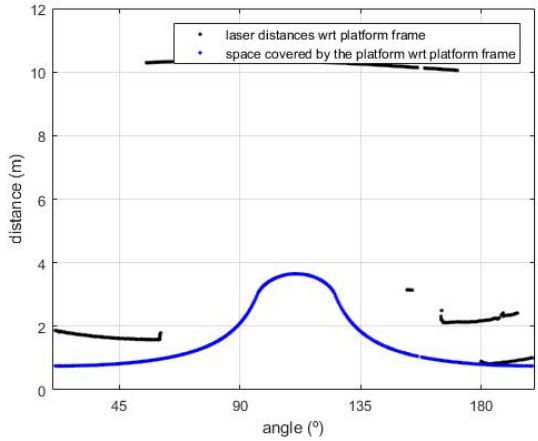

Figure 4.24: *Case 1 diagram distance vs angle.*

This case happen if according to the laser measures the robot could move with a simple translation towards the goal without having collision. We say "*could move*" because we have to remember that the filter n.2 makes not exactly thru the data collected. Particularly the data are thru until a length of $distance\_looked$ meters from the obstacle (in our work $distance\_looked = 3m$). As consequence the robot could move translating directly to the goal because it see free but during its motion the environment should change (PUT AN IMAGE OF THIS). The check to know if the robot can move directly to the goal is done comparing the distances computed using the parameterization with the distances collected with the first node (*lidar.cpp*)

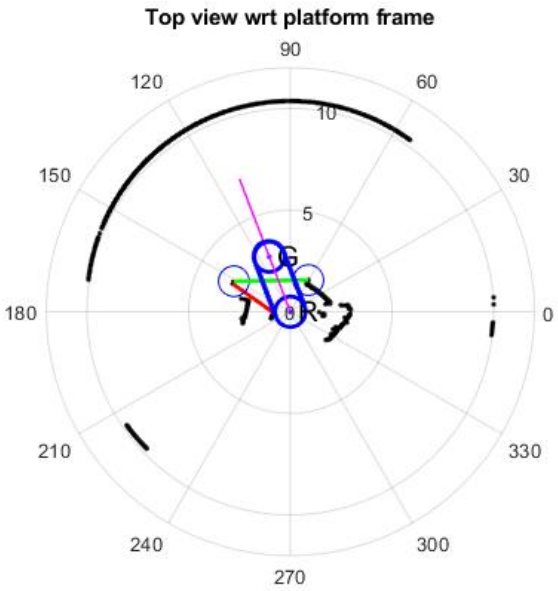

Figure 4.25: *Example of case 1, top view.*

getting and filtering the laser data. The robot can move directly to the goal if all the distances computed using the parameterization are smaller then the distances from the laser data (Figure 4.25 and Figure 4.24).

Case 2: goal on the left of the gap

The case 2 happens when the goal is located on the left of the gap (Figure 4.26). This sentence can be translated in math condition as follow:

$$IF \ \ (\hat{k} \cdot (\vec{G} \times \vec{L_{tan}}) <= 0 \ \ \&\& \ \ \hat{k} \cdot (\vec{G} \times \vec{R_{tan}}) <= 0) \rightarrow Case \ \ 2 \qquad (4.28)$$

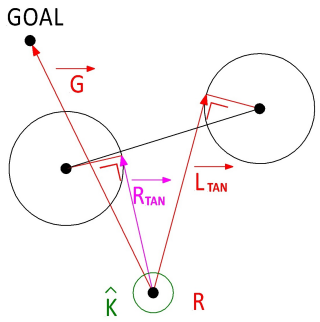

Figure 4.26: *Case 2.*

The most convenient decision to choose seems to be the direction of the vector $\vec{R_{tan}}$. Nevertheless, there are some special situation in which this choice should led to a collision. Some example in Figure 4.27.

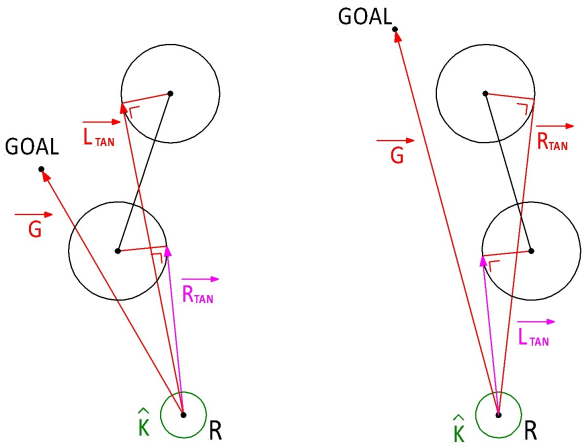

Figure 4.27: *Case 2, special situation: gap very inclined.*

Looking Figure 4.27 we can see that the situation happen when the gap is so inclined to change the reciprocal direction of the vectors $\vec{R_{tan}}$ and $\vec{L_{tan}}$. In fact in Figure 4.26 it happens $\vec{R_{tan}} \times \vec{L_{tan}} < 0$; on the contrary on Figure 4.27 it happens $\vec{R_{tan}} \times \vec{L_{tan}} > 0$. Therefore, inside the previous the condition in (4.28) it is necessary to add the following conditions:

$$\begin{aligned} &IF \ \ (\hat{k} \cdot (\vec{R_{tan}} \times \vec{L_{tan}}) >= 0)\{ \\ &\ IF \ \ (|\vec{L_{tan}}| >= |\vec{R_{tan}}|) \\ &\quad \hat{dir} = \frac{\vec{R_{tan}}}{|\vec{R_{tan}}|} \\ &\ ELSE \\ &\quad \hat{dir} = \frac{\vec{L_{tan}}}{|\vec{L_{tan}}|} \\ &\} \end{aligned} \tag{4.29}$$

With this added conditions it is possible to not collide avoiding at first the nearest border of the gap.

Case 3: goal on the right of the gap

The case 3 happens when the goal is located on the right of the gap (Figure ...). This sentence can be translated in math condition as follow:

$$IF \ \ (\hat{k} \cdot (\vec{G} \times \vec{L_{tan}}) >= 0 \ \ \&\& \ \ \hat{k} \cdot (\vec{G} \times \vec{R_{tan}}) >= 0) \rightarrow Case \ \ 3 \tag{4.30}$$

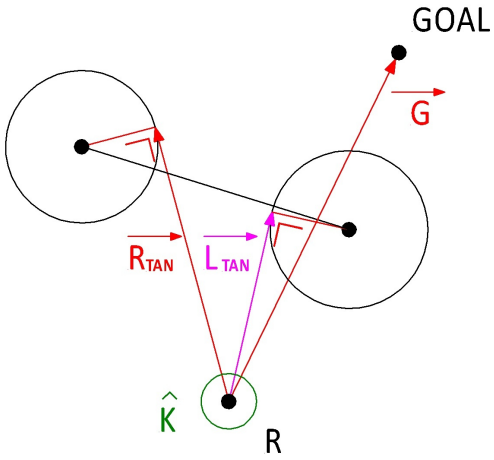

Figure 4.28: *Case 3.*

Nevertheless, similar to the case 2, there are some special situation in which this choice should led to a collision. An example in Figure 4.29.

$$\begin{aligned}
&IF \ \ (\hat{k} \cdot (\vec{R_{tan}} \times \vec{L_{tan}}) >= 0)\{ \\
&\ IF \ \ (|\vec{L_{tan}}| >= |\vec{R_{tan}}|) \\
&\quad \hat{dir} = \frac{\vec{R_{tan}}}{|\vec{R_{tan}}|} \\
&\ ELSE \\
&\quad \hat{dir} = \frac{\vec{L_{tan}}}{|\vec{L_{tan}}|} \\
&\}
\end{aligned} \tag{4.31}$$

With this added conditions it is possible to not collide avoiding at first the nearest border of the gap with the same logic used for the case 2.

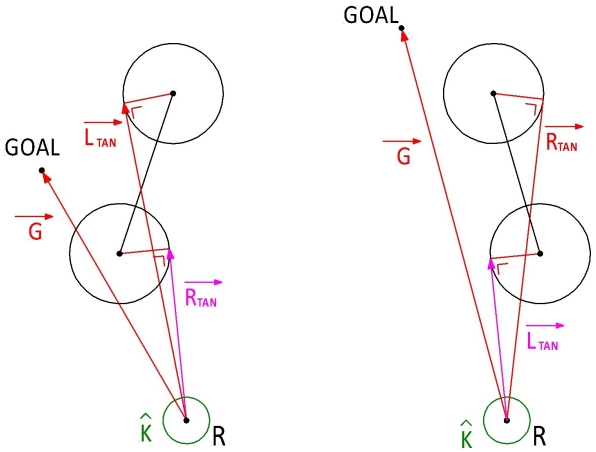

Figure 4.29: *Case 3, special situation: gap very inclined.*

Case 4: goal in the middle

The case 4 happens when the goal is located in the middle of the gap (Figure 4.30) that is when:

$$IF \ \ (\hat{k} \cdot (\vec{G} \times \vec{L_{tan}}) <= 0 \ \ \&\& \ \ \hat{k} \cdot (\vec{G} \times \vec{R_{tan}}) >= 0) \rightarrow Case \ \ 4 \qquad (4.32)$$

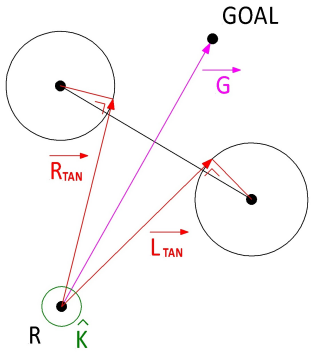

Figure 4.30: *Case 4.*

Also in this case could be a particular situation which we can see in Figure 4.31

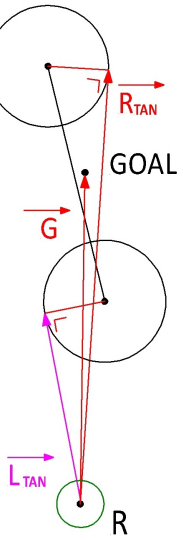

Figure 4.31: *Case 4 special.*

Case 5: no gaps detected

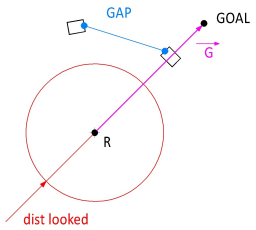

Figure 4.32: *Case 5.*

If no gaps are detected the robot it is decided (remember: no gaps detected considering only the *distance_looked*) to command the robot to go directly to the goal. Maybe during the motion the robot will detect a gap to follow otherwise it will avoid an obstacle using the logic to move orthogonal.

4.10 Set velocity

This box receives as input the unit vector $\vec{dir}$ and provides as output the vector of the generalized velocity $\vec{\mathcal{V}} = [\dot{x},\ \dot{y},\ \dot{\varphi}]^T$.

Translational velocity $\dot{x}, \dot{y}$

The setting of the translational velocity is very simple because the direction of motion is already computed $\hat{dir} = [dir_x,\ dir_y]^T$. So the velocities in the direction x and y are computed multiplying with a scale factor the component of the unit vector $\hat{dir}$:

$$\begin{aligned} \dot{x} &= scaleFactor \cdot dir_x \\ \dot{y} &= scaleFactor \cdot dir_y \end{aligned} \tag{4.33}$$

So the *scaleFactor* is set equal to the desired velocity. In our case it is set to 0,3 m/s.

Rotational velocity $\dot{\theta}$

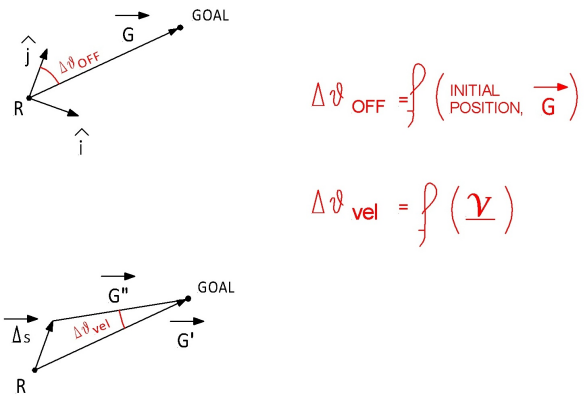

Figure 4.33: *Orientation errors.*

It is decided to want that the robot always looks to the goal during the motion. For this is applied a control for the robot's orientation in order to compute a correct value of the rotational velocity around the z-axis. The

computing is done considering two kind of angular errors in orientation as explained below.

Offset error $\Delta\theta_{OFFSET}$

It is the error due by the different between the initial orientation of unit vector $\hat{j}$ of the robot frame and the vector $\vec{G}$ wrt the robot frame. This is a constant error which depends only by the initial generalized position of the robot and the vector $\vec{G}$. So it will be recover during the motion towards the goal but it is constant during the motion itself. We can summarize this consideration with the following equation:

$$\Delta\theta_{OFFSET} = f(\hat{j}, \vec{G}) = acos\left(\frac{\vec{G}}{G} \cdot \hat{j}\right) \tag{4.34}$$

It is assume to recover the offset error in $Time = 5$ seconds. Then it is done a discretization considering the ROS frequency set. Particularly, it is set $frequency = 100$ Hz. This means that every $1/100$ sec the program are executed and a velocity command to the platform is sent. In order to sent a reasonable rotational velocity to the platform are applied the following procedure:

1. compute of the number of commands sent in the $Time$ set:

$$numCommands = \frac{Time}{1/freq} = \frac{Time}{\Delta t} \tag{4.35}$$

2. do the discretization computing the step angle to recover $\Delta\theta$ sending one command to the platform:

$$\Delta\theta = \frac{\Delta\theta_{OFFSET}}{numCommands} \tag{4.36}$$

3. compute the rotational velocity value in magnitude and sign; the sign is determined considering positive a counterclockwise rotation (according to the right hand convention around the z-axis) and negative for a clockwise rotation. In order to choose the correct sign is compute the *sin* of the angle between the direction given by $\hat{j}$ and $\vec{G}$. In the code

this is implemented in this way:

$$\begin{aligned} &IF \ \left(\hat{k} \cdot \left(\frac{\vec{G}}{G} \times \hat{j}\right) < 0\right) \\ &\dot{\theta}_{OFFSET} = +\frac{\Delta\theta_{OFFSET}}{1/freq} \\ &ELSE \\ &\dot{\theta}_{OFFSET} = -\frac{\Delta\theta_{OFFSET}}{1/freq} \end{aligned} \tag{4.37}$$

The angle $\Delta\theta_{OFFSET}$ will be recover when the number of commands sent are exactly equal to $numCommands$. For this reason in the code a counter *count* variable is used. It is incremented each time that the algorithm is run and when $count = numCommands$ the rotational velocity $\dot{\theta}_{OFFSET}$ is reset to zero.

Error due by the translational velocity: $\Delta\theta_{VEL}$

The translational velocity of the platform results in a misalignment between the vectors $\hat{j}$ and $\vec{G}$. We call this error due by translational velocity $\Delta\theta_{VEL}$.

In the Figure 4.33 is showed the vectorial equation used in order to compute $\Delta\theta_{VEL}$:

$$\vec{\Delta s} + \vec{G''} = \vec{G'} \tag{4.38}$$

Splitting the (4.38) in the direction of the vector $\vec{G'}$ and the orthogonal one we can write the following equation:

$$\begin{aligned} &\rightarrow \Delta s \cdot sin(\alpha) = G'' \cdot sin(\Delta\theta_{VEL}) \\ &\uparrow \Delta s \cdot cos(\alpha) + G'' \cdot cos(\Delta\theta_{VEL}) = G' \end{aligned} \tag{4.39}$$

The vector Δs is a finite displacement which happens during the time $\Delta t = 1/freq$ cause by the translational velocity:

$$\Delta s = \sqrt{\dot{x}^2 + \dot{y}^2} \tag{4.40}$$

The vector $\vec{G'}$ is the distance from the goal in the current instant t. The vector $\vec{G''}$ is the distance from the goal in the instant next $t + \Delta t$, where Δ. It is given by:

$$G'' = \sqrt{G'^2 + \Delta s^2 - 2 \cdot G' \Delta s \cdot cos(\alpha)} \tag{4.41}$$

Solving the system in (4.39) we can compute $\Delta\theta_{VEL}$:

$$\Delta\theta_{VEL} = asin\left(\frac{\Delta s \cdot sin(\alpha)}{G''}\right) \tag{4.42}$$

Moreover, we define for convenience the unit vectors:

$$\begin{aligned} \hat{\Delta s} &= \frac{\vec{\Delta s}}{\Delta s} \\ \hat{G} &= \frac{\vec{G}}{G} \end{aligned} \tag{4.43}$$

The function *asin* in C++ provides a value always positive so, in order to choose the correct sign, is applied a logic similar to the logic used in (4.37):

$$\begin{aligned} &IF \ \left(\hat{k} \cdot (\hat{\Delta s} \times \hat{G}) \geq +\epsilon\right) \\ &\dot{\theta}_{VEL} = +\frac{\Delta\theta_{VEL}}{\Delta t} \\ &ELSE \ IF \ \left(\hat{k} \cdot (\hat{\Delta s} \times \hat{G}) \leq -\epsilon\right) \\ &\dot{\theta}_{VEL} = -\frac{\Delta\theta_{VEL}}{\Delta t} \\ &ELSE \ IF \ \left(-\epsilon \leq \hat{k} \cdot (\hat{\Delta s} \times \hat{G}) \leq +\epsilon\right) \\ &\dot{\theta}_{VEL} = 0 \end{aligned} \tag{4.44}$$

In (4.44) is introduce the tolerance ϵ. It is necessary because during the motion the unit vectors $\hat{\Delta s}$ and $\hat{G}$ could be aligned but only for an instant. The tolerance is expressed in term of *sin* of the *angle tolerance* δ as below:

$$\epsilon = sin(\delta) \tag{4.45}$$

Particularly, in our case the *angle tolerance* is set to: $\delta = 5° \cdot \pi/180$.

Moreover, it is set a proportional control for the translational velocity which is activated when the robot is close to an obstacle with $\vec{vec_{MIN}} < dist_1$. In other words we can imagine to have two zones:

- $dist_2 < \vec{vec_{MIN}} < dist_1$ In this zone the control proportional of the velocity is done changing the scale factor with the following relation:

$$\begin{aligned} scaleFactor &= \frac{|\vec{vec_{MIN}}|}{dist} \cdot vel_{MAX} \\ IF \ &(scaleFactor < vel_{MIN}) \\ &scaleFactor = vel_{MIN} \end{aligned} \tag{4.46}$$

The last IF allows to avoid the situation which

- $|\vec{vec_{MIN}}| \geq dist_1$ In this zone, that is a safety zone, the robot can move at the max velocity set:

$$scaleFactor = vel_{MAX} \tag{4.47}$$

- $|\vec{vec_{MIN}}| < dist_2$ It is a dangerous zone. So in this case it is decided to stop the robot to avoid any not safety movement of the robot because it is so much close to obstacles and it could collide with them:

$$scaleFactor = 0 \tag{4.48}$$

This logic allows to avoid obstacles with a smallest translational velocity.

A similar logic is used to set the translational velocity when the robot is approaching to the goal that is when $G < dist_{approach}$.

$$scaleFactor = \frac{G}{dist_{approach}} \cdot vel_{MAX} \tag{4.49}$$

it is also set a distance to say if the robot is almost arrived to the goal or not. The tolerance set for the stop distance, distance from the goal is:

$$\begin{gathered} IF \ \ (G < dist_{GOAL}) \\ scaleFactor = 0 \end{gathered} \tag{4.50}$$

5. Experimentations

As it is explained in the previous chapter, thanks to the algorithm it is possible to get concrete data about the gaps, choose the best gap and the best direction to follow. In the following picture we will give some examples of the experiment done and the results obtained showing some pictures referred to a specific instants during the motion. Particularly, it will be showed some Rviz pictures (which allow to visualize the real laser signals) and some Matlab pictures (in which it is possible to see the results of the post-processing).

5.1 Experiment 1

In this first example will we showed the results of the algorithm with the filter n.2 applied with a distance looked = 3 m (Figure 5.1) and with distance looked = 6 m (Figure 5.2) in order to see the difference. In fact, after the experimentation we understood that for the algorithm proposed it is not always useful to consider the obstacle very far to the robot. It is better to consider only the obstacles eventually located near to the robot, in concrete term it is useful to set the parameter *distance_looked* on 3 or 4 meters around it. The reason it is understandable comparing Figure 5.1 and Figure 5.2. Both of this Figure are referred to the some situation shown in the picture on the right of Rviz. We can see that in the case in which the parameter distance looked = 6 m (Figure 5.2) the gaps built (in red and green for the best to follow) are different that the ones built with a distance looked = 3 m (Figure 5.1). Particularly, in the case with larger distance looked the gaps detected are less useful then the gaps built in the case with smaller distance looked. This is due by different width of the jumps in the laser measures. So, it is believed that it is better to set a parameter distance looked shorter. Moreover, it is believed that the correct value of this parameter should be chosen considering the dimensions of the room the robot navigate in and the density of the obstacle which could appear. If the density of the obstacle is high it should be convenient to choose a smaller value of the parameter. In the Figure 5.1, Figure 5.3, Figure 5.5 it is possible to follow the sequence of the algorithm results in four different instants during the motion of the robot.

5.2 Experiment 2

In this example it is possible to see the state machine in the case0. In fact, the purple line in Figure 5.7 is orthogonal to the $\vec{vec}_{MIN}$. This vector is not drawn in the picture but it is easy to imagine that it comes from the robot R and arrives to one point of the close obstacle in black.

5.3 Experiment 3

In this experiment is showed how a long obstacle (long with respect to the dimension of the platform) is very close to the platform could led to a failure of the algorithm (Figure 5.9). What happens is that the robot starts to move in orthogonal direction, for example on the right, but after some instants it will move on the left and so on as a loop without reaching the goal. On the contrary, if the obstacle is a little bit far to the robot, the latter should detect some gaps on the boundaries of the obstacle and avoid it (Figure 5.9).

5.4 Comments on the parameterization

In Figure 5.4 and Figure 5.6 are showed the diagram related to the parameterization for two different instants of the robot motion. We want to focus the attention on the blu and black lines in these pictures. As it is explained in the previous chapter, the algorithm decided to choose a direction of the motion aligned with the vector $\vec{G}$ (that means we are in the case 1 of the state machine: move the robot directly to the goal) if the blu line is under the black one. This means that moving directly to the goal the robot does not meet obstacles. It is the case showed in Figure 5.6. On the contrary if the black line is under (even in one point) the blu one, the algorithm provide to select the correct *case* of the state machine in order to not provide collision in the motion. It is the case showed in Figure 5.4.

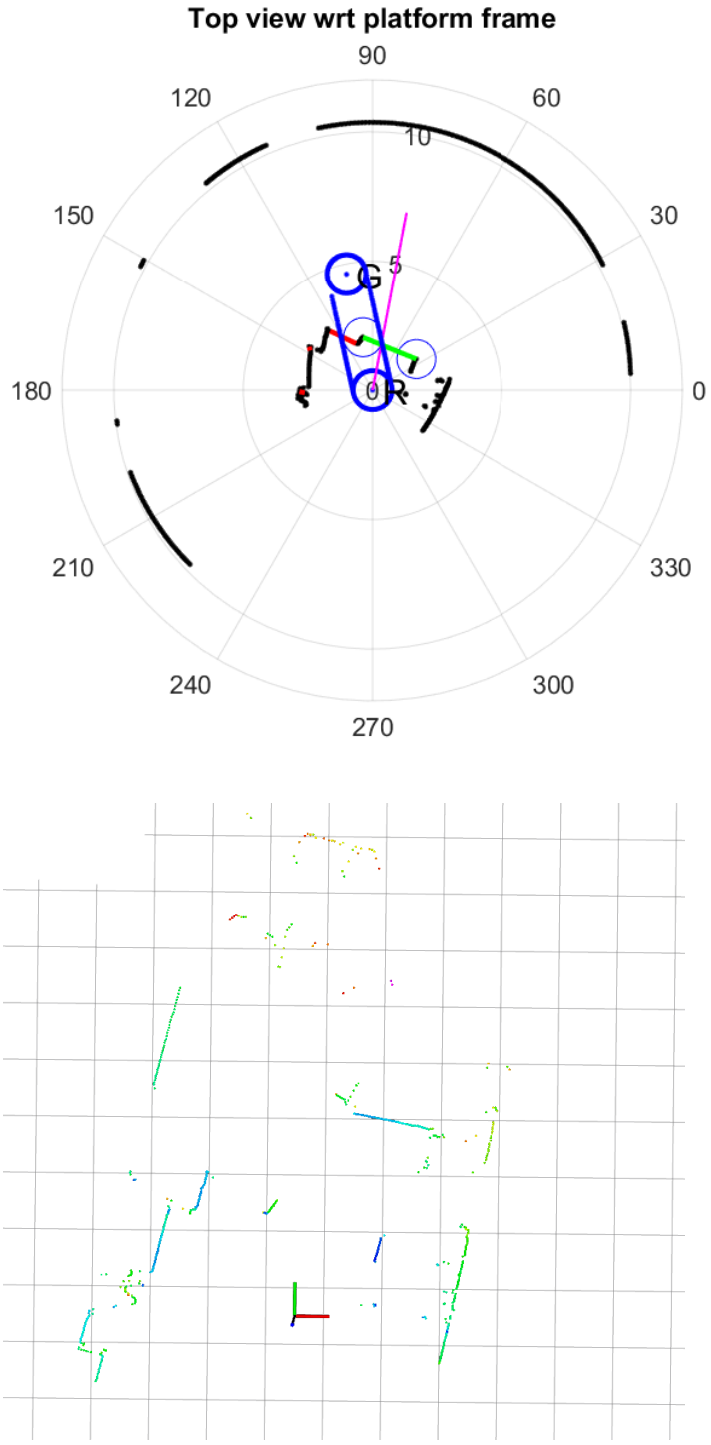

Figure 5.1: ***Experiment 1****: Top view Matlab, instant 1, with distance looked = 3 m. State machine: case2.*

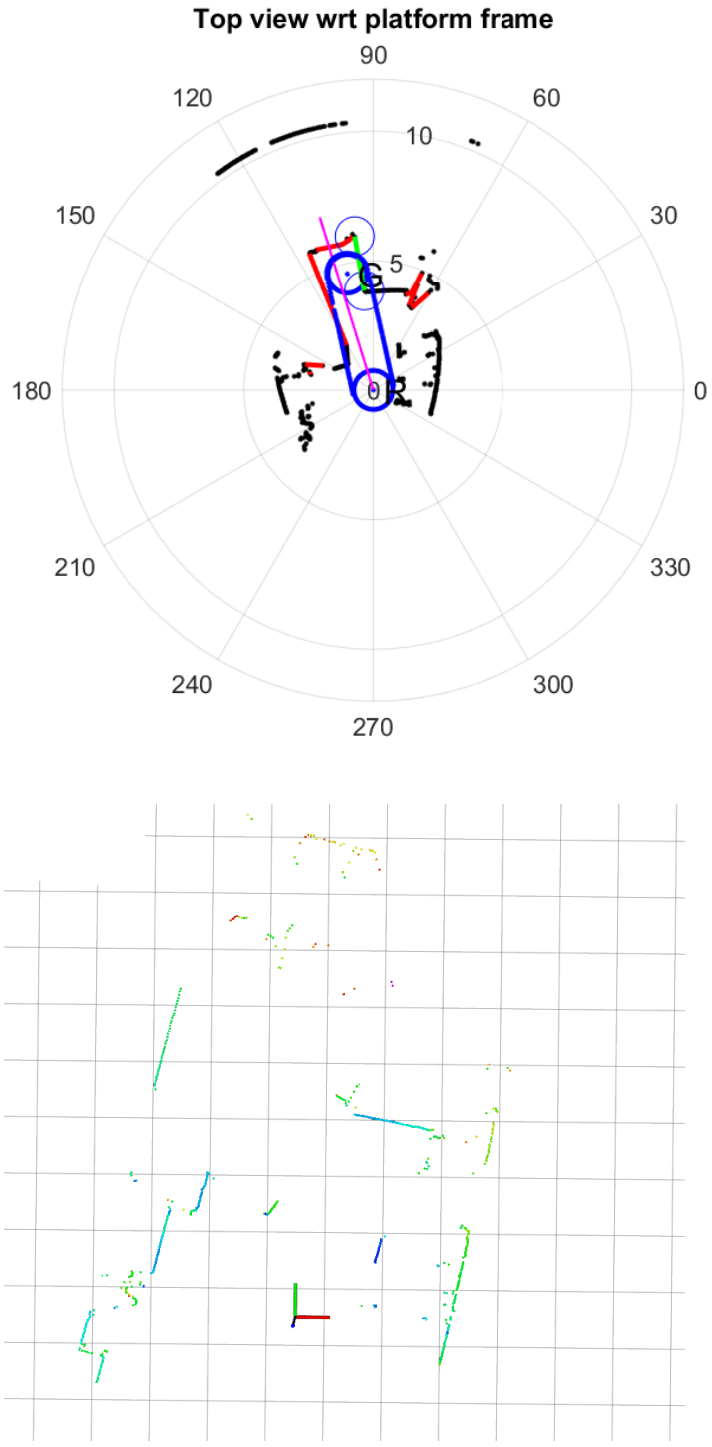

Figure 5.2: ***Experiment 1****: Top view Matlab, instant 1, with distance looked = 6 m. State machine: case3 special.*

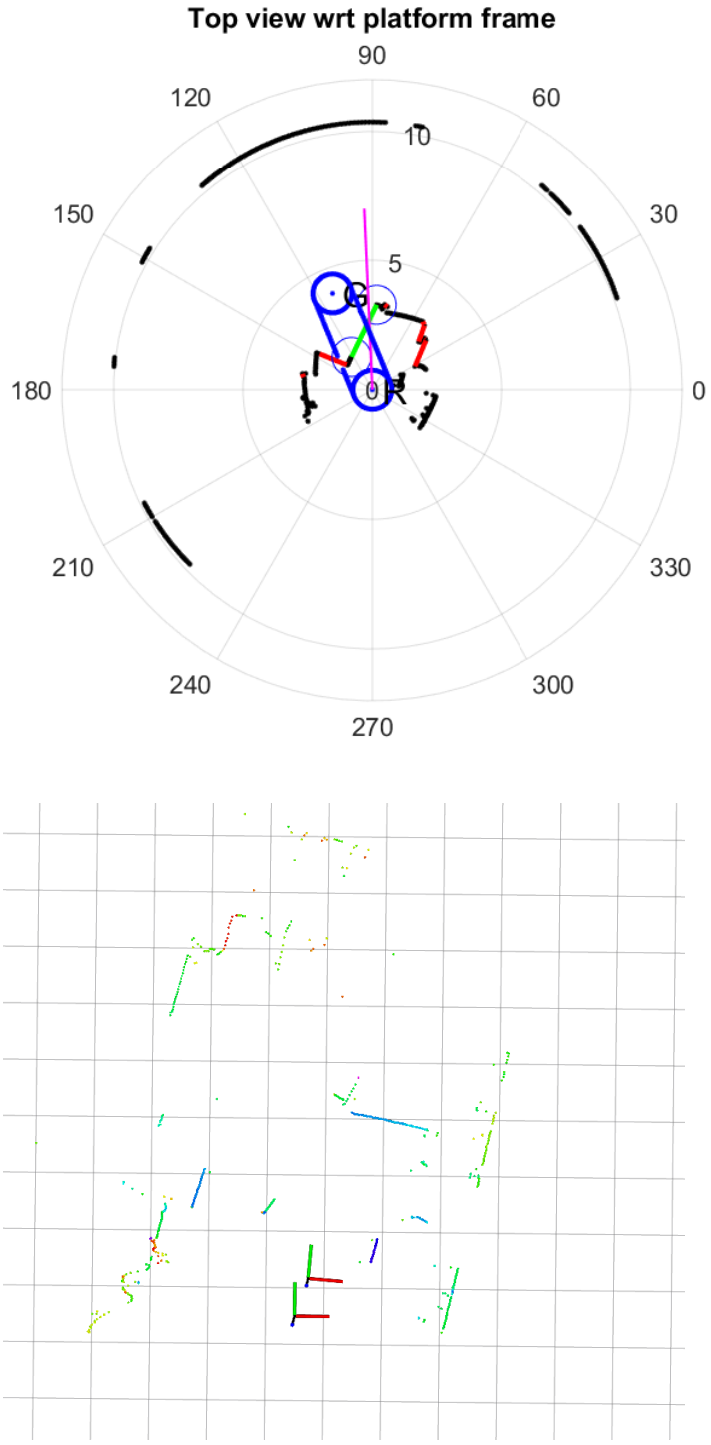

Figure 5.3: ***Experiment 1****: Top view Matlab, instant 2, with distance looked = 3 m. State machine: case2.*

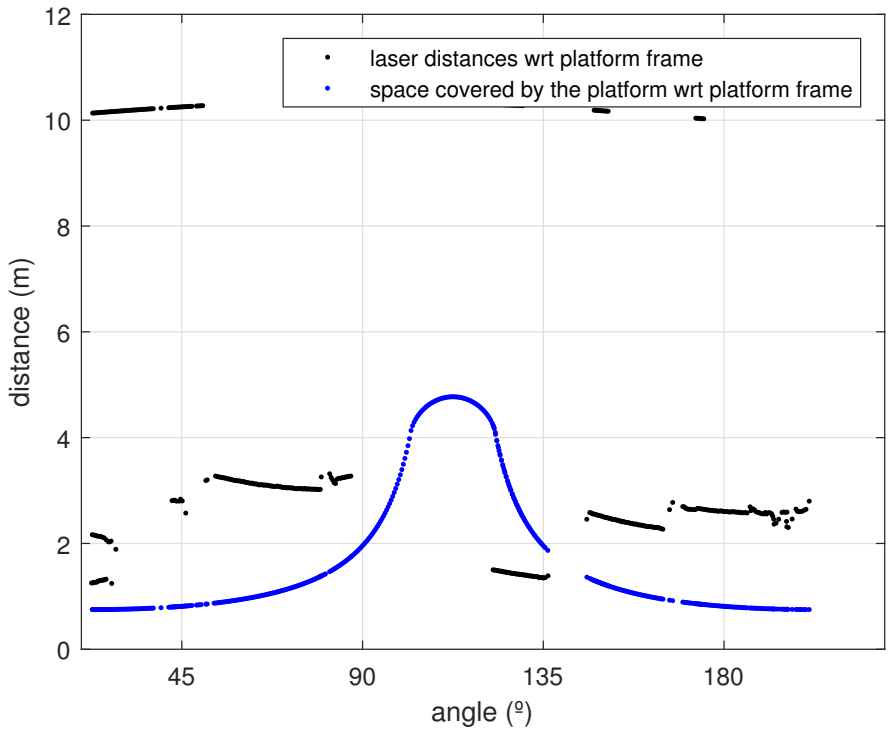

Figure 5.4: *Parameterization in showed in the diagram angle(°) vs distance(m). The point in black are the points of the distances detected by the laser (with the modification doing by the filtering); the point in blu are the points done by the parameterization.*

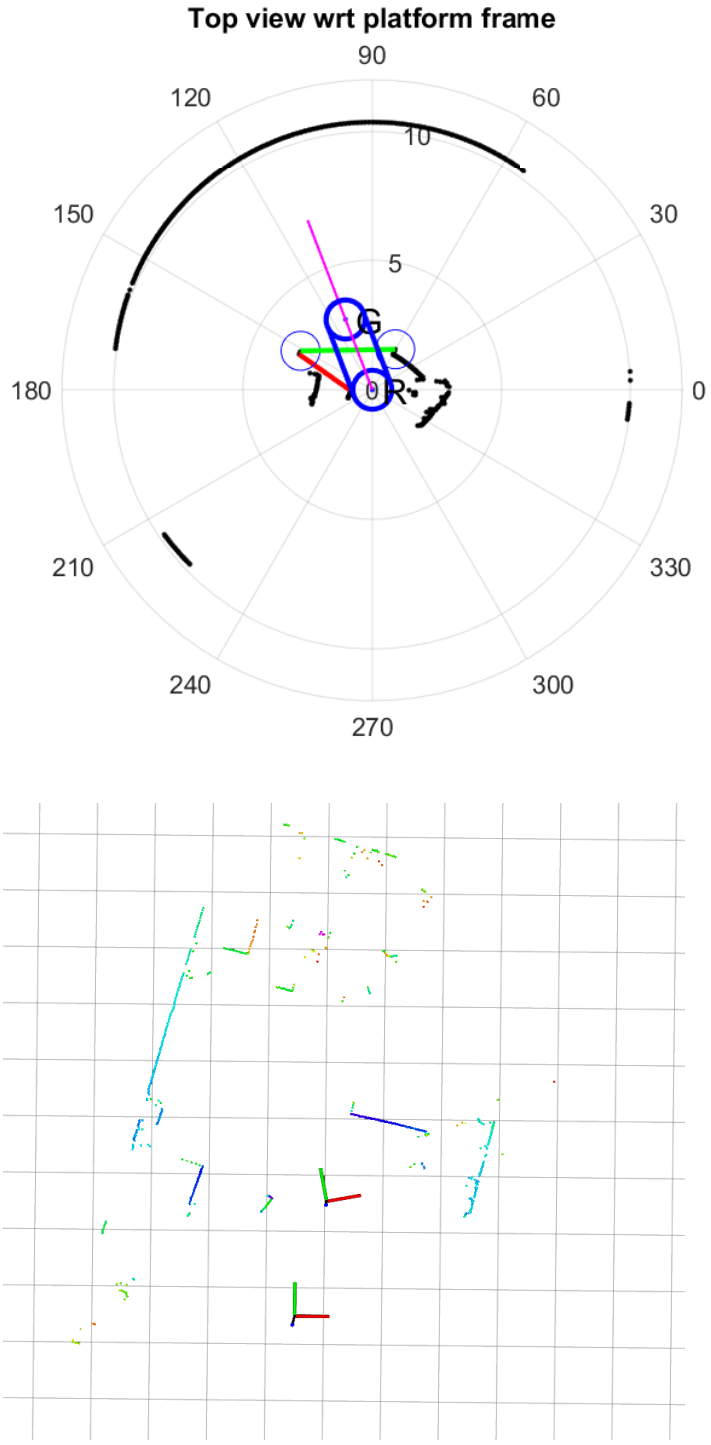

Figure 5.5: ***Experiment 1****: Top view Matlab, instant 3, with distance looked = 3 m. State machine: case5.*

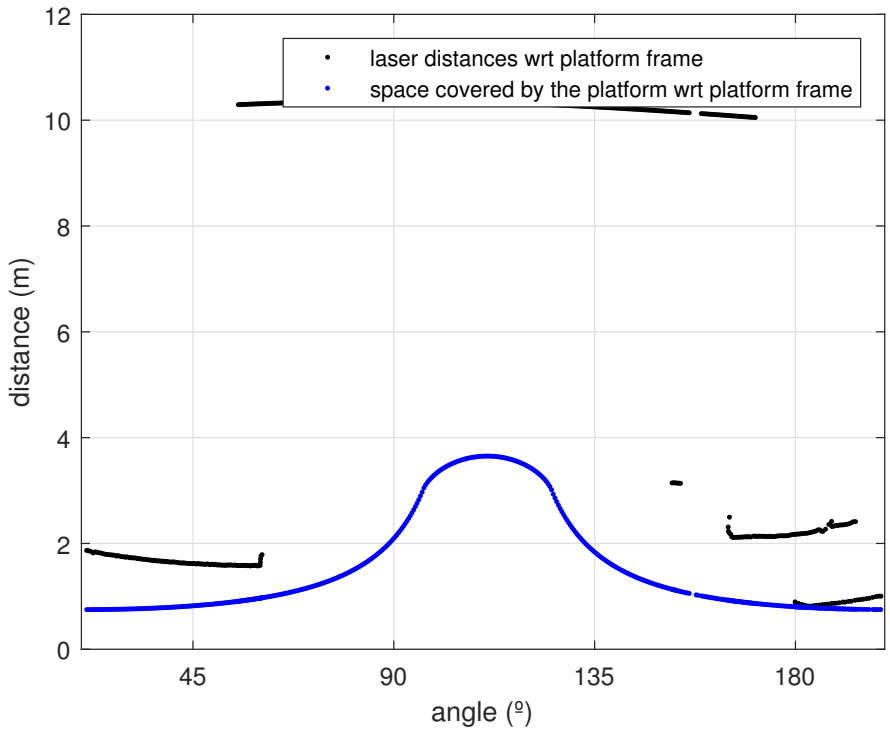

Figure 5.6: *Parameterization in showed in the diagram angle(°) vs distance(m). The point in black are the points of the distances detected by the laser (with the modification doing by the filtering); the point in blu are the points done by the parameterization.*

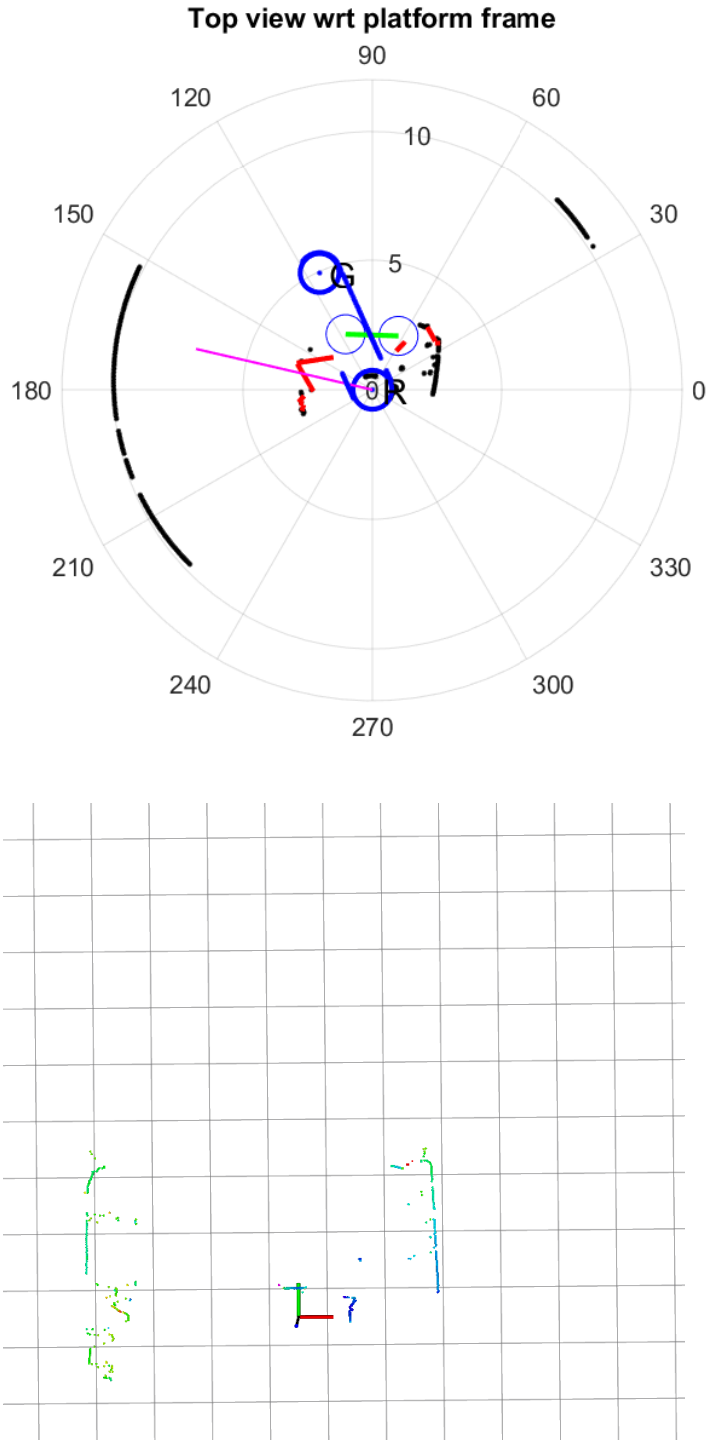

Figure 5.7: ***Experiment 2****: Top view Matlab, instant 1, with distance looked = 3 m. State machine: case0.*

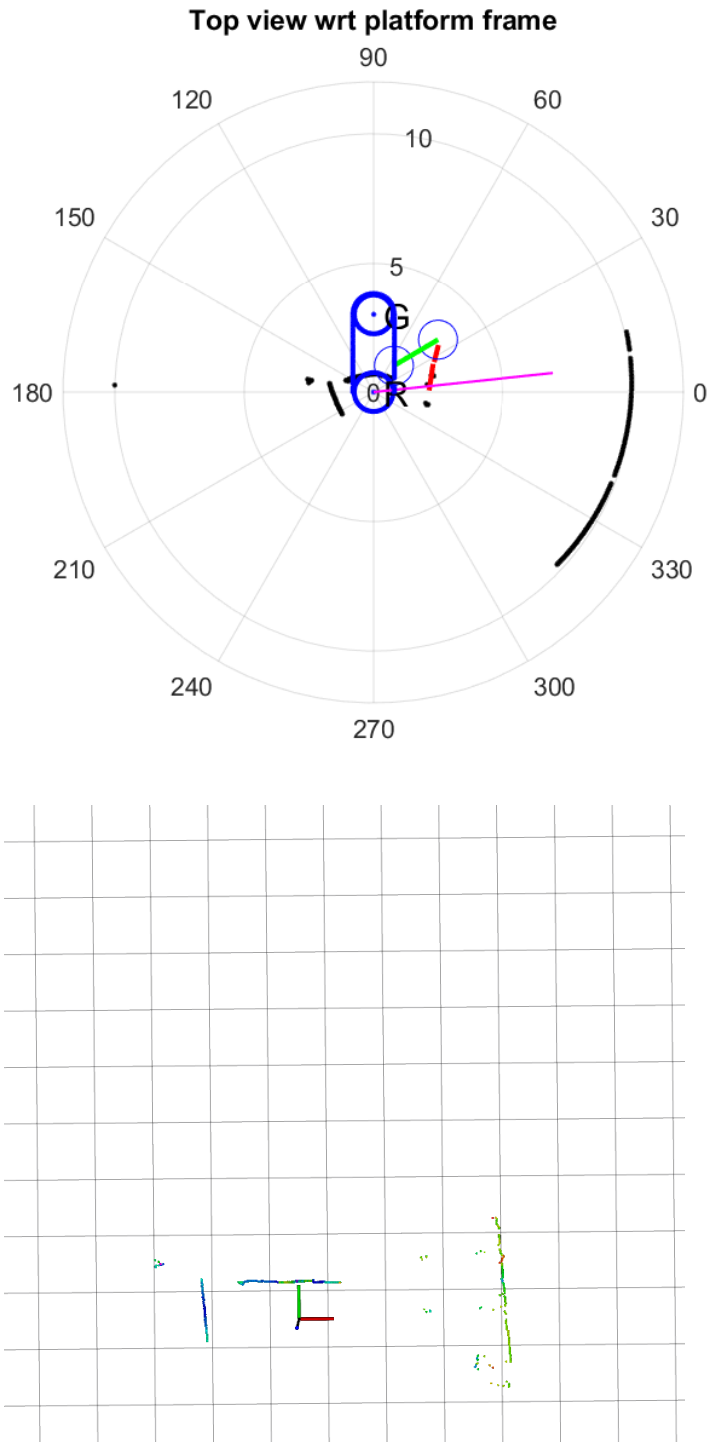

Figure 5.8: ***Experiment 3****: long obstacle very close to the robot, with distance looked = 3 m. State machine: case0.*

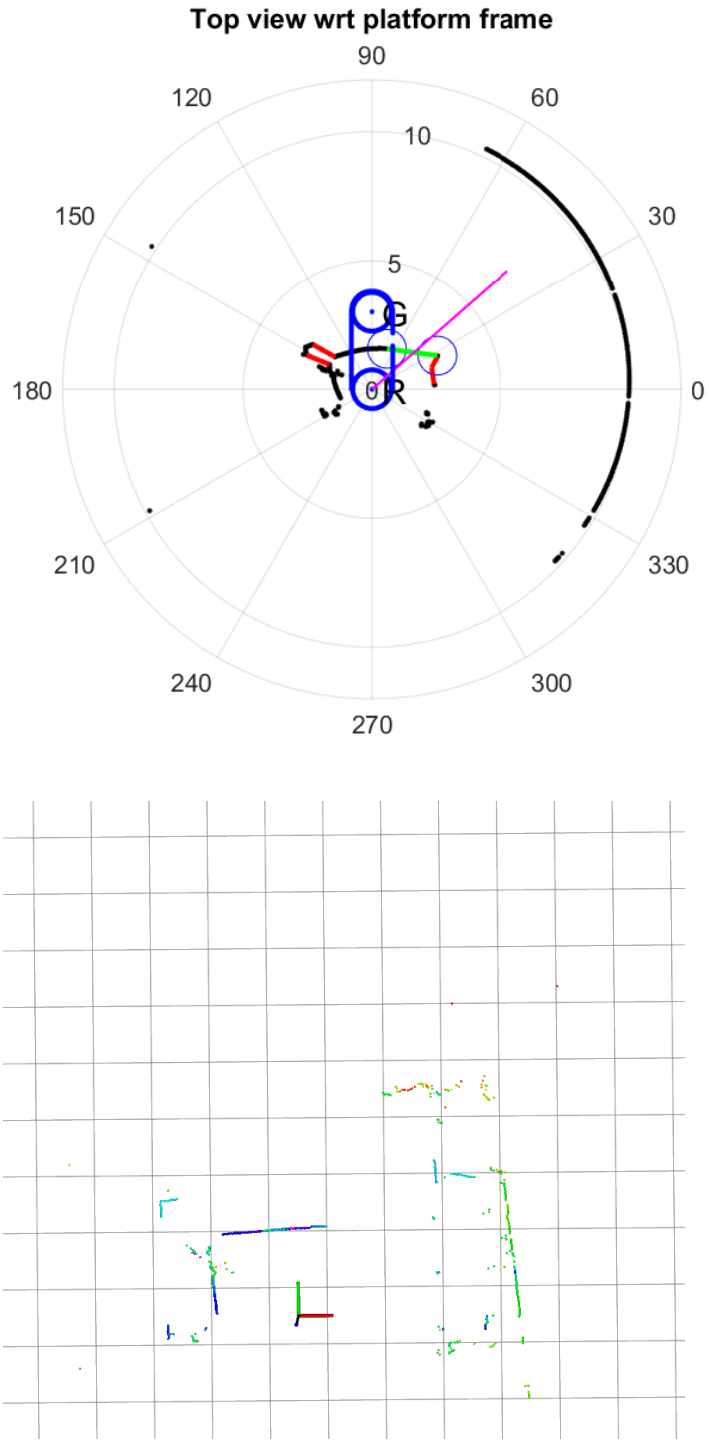

Figure 5.9: ***Experiment 3****: long obstacle not very close to the robot, with distance looked = 3 m. State machine: case1.*

6. Comparison with other avoidance technique

Several obstacle avoidance algorithm have been proposed. In this chapter we want to compare some of this with the algorithm proposed in this work in order to understand its pros and cons.

6.1 Bug Algorithm

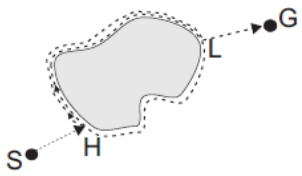

Figure 6.1: *Bug algorithm.*

The Bug algorithm (Figure 6.1) is the simplest algorithm which has proposed. It consists in fully circle the obstacle in order to find the point with shortest distance to the goal. Then the robot will circle again the object leaving the boundary of the obstacle from this point and going to the goal. This algorithm very inefficient was improved and a second version was proposed named *Bug2*.

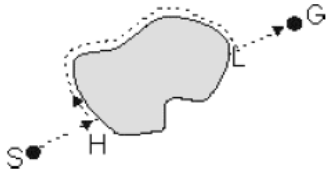

Figure 6.2: *Bug2 algorithm.*

With the *Bug2* the robot starts to circle the obstacle but leaves the boundary when its position intersects the line which connects the robots starting point and the goal.

6.2 Potential fields method

It is an elegant approach to solved the problem. It consists to imagine the robot, the obstacles and the goal as electric charges. The obstacles are charges with the same sign of the robot; the goal is a charge with opposite sign with respect to the robot. So, the potential field algorithm assumes that the robot is driven by virtual forces that attract it towards the goal, or reject it away from the obstacles. The actual path is determined by the resultant of these virtual forces (Figure 6.3).

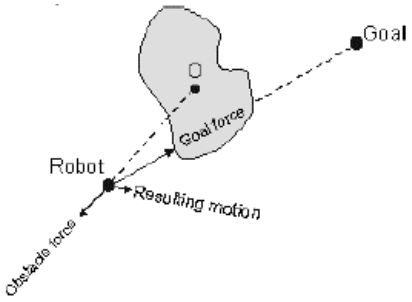

Figure 6.3: *Potential fields method.*

6.3 Approach proposed in this work

The approach proposed in this work is more efficient of both the methods previously explained because the robot can move towards the goal going directly through the gaps detected. This allows the robot to start to avoid the obstacle at a consistent distance from it. As consequence, the robot will cover a shorter path comparing to that it would cover using the previous methods.

7. Costs and environmental impact

7.1 Costs

The economic cost of the project has to consider: the depreciation related with the project of the robotics equipment and computers of the laboratory, the costs of the working hours of the person involved (students and supervisors), and the electrical energy consumed during the experiment as well as the development of the algorithm solution.
The robotics equipment used in this project is only the prototype robot MADAR made in IOC. I is estimated that MADAR has a useful life of 10 years. Considering the use of MADAR of 6 hours a day, 5 days a week, 20 weeks (more or less 5 months), it is estimated a useful life of the robot in hours of 6000 hours. Two computers are used in the lab (one remote and one on board). They have an estimated useful life of 5 years. Using them an average time of 10 hours a day, it is estimated a useful life of 1200 hours. Depreciation can be then calculated from the total price of the robot and the computers, the knowledge of their useful life and the total time that they have been used. The project has been completed from February to June with and average of 4 hours per day. This generates and approximate total time of 400 hours invested in development and testing. The two computers of the lab have been running constantly for almost all of these time so a percentage of 95% will be estimated for them, which means a total of 380 hours of use. The working hours of the student can be estimated of 400 hours employed in the lab to complete project plus 20 hours more spent in writing this report and completing other documentation tasks. This is a total of 420 hours spent. As ETSEIB recommends, the salary for the students will be considered of 8 euro/h. Supervision and meeting hours with the director of the project and other members of the staff of the laboratory will be considered of 50 hours in total with an average cost of 30 euro/h. Finally, the electrical cost can be summarized by only considering the energy consumed by the computers and the robot during their working hours. The rest of electrical elements of

the room are not considered. With an average electrical cost in 2019 of 0,15 euro/kWh, the consumption of both computers can be estimated to be of about 0,40 kWh. MADAR, on the other hand, has an electrical consumption of 0,72 kWh. Next table presents all the costs described and analyzed. The total cost of the project finally is of 9789,75 euro.

7.2 Environment impact

In the particular field of this project, service robotics, the impact is much lower. In fact, using this kind of mobile manipulators in many different business could incur in an increment on energy consumption by this establishments. But by just analyzing the project and the solutions that might come from it and be developed around it, it does not seem very relevant to think that it might have any environmental impact. The relevance of this project from the environmental impact point of view is almost null.

8. Future work

As it is explained the algorithm developed uses only the front laser scanner. So, one of the first thing to do in order to improve it is to integrate the use of the other two laser available on the platform. In order to do this it needs to consider that there are some area sectors which are covered at the same time by more then a laser scanner and sector that are covered only by one of the laser. It is not immediate to decide a strategy to manage the information taken from three lasers.
A possible approach could be to try to get the boundaries splitting the data of the three laser scanners. Then built, for each of them the gaps and finally try to make a *sensor fusion* of that data. After this the algorithm to choose the best direction knowing the gaps could be applied.
Another important problem to solved is the noise is to try to delete the noise of the laser with some kind of filtering technique.
Finally, the use of RGB-D camera of which the robot is equipped and the radar positioning system available in the lab could increase the knowledge of the environment around the robot. Obviously, the manage of the data will be more complex.

Acknowledgements

I would like to thank some people.

Firstly, I thank my family that supported me economically and morally, and gave me strenght and courage to overcome every difficulties during my studies. I owe my achievement to them.

I am thankful to my supervisor Raúl Suárez who gave me the opportunity to develop my thesis in a stimulating environment that is the Institute of Industrial and Control Engineering (IOC) at the Universidad Politecnica de Catalunya (UPC). I am very grateful to Leo Palomo and the Phd student Andrés Felipe Montaño who was the person that helped me more during the develop of my work.

Moreover, I can not omit to thank many friends knew during my years at university for all the happy memories share together.
I would like to express my gratitude to the friends of mine from Collegio Einaudi and particularly to Francesco, Roberto, Riccardo, Angelo, Raffaele, Sergio, Simone, Rita, Vincenzo.
Least but not last, I thank my childhood friends and the ones encountered during the awesome Erasmus experience in Barcelona with which I kept in touch daily despite of the distance.

A. Script

Listing A.1: Script of *lidar node*.

```
1
2  #include <sensor_msgs/LaserScan.h>
3  #include <ros/ros.h>
4  #include <stdio.h>
5  #include <vector>
6  #include <math.h>
7
8  #include <madar/vectorArray.h>        // include the msg
       file !!!
9  //#include <madar/SetDistance.h>       // include
       service file !!!
10
11 #include <iostream>
12 #include <fstream>
13
14
15
16
17 madar::vectorArray vectorDataList;   // array of type
       vectorData.msg
18                                       //
                                           ____________________________


19                                       // float64 x
20                                       // float64 y
21                                       //
                                           ____________________________


22 madar::vectorData my_vectorData;
23
24 // Laser features (step angle = (degrees), radius = [m
```

```
    ], max laser distance [m]
// step_angle_laser ---> proportion 270 : msg.ranges.
    size() =  step_angle : 1
// msg.ranges.size() = 686 ---> step_angle_laser =
    270/686 = 0.3935
float step_angle_laser = 0.3935, distance_looked = 3;
float filter_distance = 0.005; // filtering distance
   WRT the LASER!

float auxiliar_variable, distance, angle, deg2rad=M_PI
    /180;
bool first_execution_front = true,
    first_execution_front2 = true;

bool first_cycle = true;
int number_filtered=0;
int frequency = 100;

void frontCallback(const sensor_msgs::LaserScan &msg) {
    vectorDataList.array.resize(0);
    int j = 0;
    for (int i = 0; i < msg.ranges.size(); i++){
//        std::cout << "SIZE MSG " << msg.ranges.size()
    << std::endl;

        // build distance vector wrt laser frame
        angle = i*step_angle_laser-45; // (degrees)

        if(angle<0)
            angle=angle+360; // (degrees);


        if((msg.ranges[i] > distance_looked) || (isinf(
           msg.ranges[i])))
            distance = 10;
        else
            distance = msg.ranges[i];

      // FILTERING
      if (distance >= filter_distance) {

```

```
58          my_vectorData.x = distance * sin(deg2rad *
                angle);
59          my_vectorData.y = - distance * cos(deg2rad *
                angle);
60
61          vectorDataList.array.push_back(my_vectorData);
62       }
63       else{
64          number_filtered++;
65 //          std::cout << "filtered" << std::endl;
66          }
67       // END FILTERING
68
69      }
70
71      first_cycle = false;
72
73
74
75      // I write the output of the node in a txt file:
76      // txt with 2 colons: x and y wrt laser frame
77
78      // x_laser, y_laser
79      // ..       ..
80      // ..       ..
81
82      if (first_execution_front){
83 //         first_execution_front=false;
84      std::ofstream myfile ("front_scan.txt");
85      if (myfile.is_open()){
86        std::cout<<"FILE OPENED" << std::endl;
87        for(int count = 0; count < vectorDataList.array.
              size(); count ++){
88           myfile <<  vectorDataList.array[count].x << "
                 " << vectorDataList.array[count].y << " "
                 << std::endl;
89        }
90        myfile.close();
91      }
92      else std::cout << "Unable to open file";
93      }
```

```
94
95
96      if (first_execution_front2){
97  //          first_execution_front2=false;
98      std::ofstream myfile ("front_scan_distance.txt");
99      if (myfile.is_open()){
100       std::cout<<"FILE OPENED" << std::endl;
101       for(int count = 0; count < vectorDataList.array.
             size(); count ++){
102           myfile <<  sqrt(pow(vectorDataList.array[
                 count].x, 2.) + pow(vectorDataList.array[
                 count].y, 2.)) << " " << std::endl;
103       }
104       myfile.close();
105     }
106     else std::cout << "Unable to open file";
107     }
108
109     std::cout << "size: " << vectorDataList.array.size
          () << std::endl;
110
111 }
112
113 //// SERVER CALLBACK
114 //bool setDistance(madar::SetDistance::Request  &req,
115 //                 madar::SetDistance::Response &res)
      {
116 //    try{
117
118 //          distance_looked = req.newdistance + 2.;
119 //          res.success = true;
120
121 //          std::cout << "distance_looked: " <<
     distance_looked << std::endl;
122
123 //          return true;
124 //    } catch (std::exception &e) {
125 //          ROS_ERROR("%s",e.what());
126 //          return false;
127 //    }
128 //}
```

```
129
130
131
132 int main(int argc, char **argv)
133 {
134     ros::init(argc,argv,"lidar");
135     ros::NodeHandle n;
136
137     ros::Subscriber sub_front = n.subscribe("/front/
          scan", 1, frontCallback);
138     ros::Publisher pub_front = n.advertise<madar::
          vectorArray>("/front_out/scan",1000);
139
140 //    ros::Subscriber sub_left = n.subscribe("/left/
    scan", 1000, leftCallback);
141 //    ros::Subscriber sub_right = n.subscribe("/right/
    scan", 1000, rightCallback);
142
143     //SERVER DECLARATION
144 //    ros::ServiceServer distance_srv = n.
    advertiseService("setDistance",setDistance);
145
146     ros::Rate rate(frequency);
147     while(ros::ok()) {
148
149         std::cout << "distance_looked: " <<
              distance_looked << std::endl;
150
151         pub_front.publish(vectorDataList);
152
153         ros::spinOnce();
154         rate.sleep();
155     }
156 }
```

Listing A.2: language=C++, caption=Script of *algorithm node*.

```
#include <ros/ros.h>
#include <stdio.h>
#include <vector>
#include <math.h>
#include <Eigen/Dense>

#include <madar/vectorArray.h>        // include the msg file !!!
#include <madar/laserArray.h>
//#include <madar/SetDistance.h>       // include service file !!!

#include <iostream>
#include <fstream>

//---------------- INCLUDE FROM XAVI (most of these are not necessary!)
#include <geometry_msgs/PoseStamped.h>
#include <geometry_msgs/Pose.h>
#include <geometry_msgs/TransformStamped.h>
#include <geometry_msgs/Twist.h>
#include <sensor_msgs/JointState.h>
#include <tf/transform_listener.h>
#include <tf/LinearMath/Matrix3x3.h>
#include <tf/LinearMath/Quaternion.h>
#include <tf/LinearMath/Vector3.h>
#include <tf2_ros/transform_listener.h>
#include <tf2_geometry_msgs/tf2_geometry_msgs.h>
#include "tf/tf.h"
#include "tf/tfMessage.h"
//-------------------------------------


// VARIABLES DECLARATION

// variables for get_boundaries
typedef struct gap {
    float left_boundary_vec[2];
    float right_boundary_vec[2];
```

```
37 }gap;
38 gap my_gap;
39 std::vector<gap> gapList;
40
41 // laser measures in some direction
42 typedef struct laser{
43     float distance;
44     float angle;
45 }laser;
46 laser my_laser;
47 std::vector<laser> laserList;
48
49
50 float radius_platform = 0.36, previous_magnitude,
      current_magnitude, jump, d_safe = 0.8, d_safe_vec
      [2];
51 float left_boundary_vec_LIM[3], right_boundary_vec_LIM
      [3];
52 float deg2rad=M_PI/180;
53 bool X = true, Y = true, first_execution_front=true,
      printCase = false;
54 bool W = true;
55 float alpha1, alpha2, alphaGoal, alpha3, alpha4, theta;
56
57 float platform_width = 1.;
58
59 // variables for "get_goal"
60 geometry_msgs::Pose goalPoseWorld;
61 geometry_msgs::PoseStamped goalWRTBaseLink;
62 Eigen::Vector3d goalWRTBaseLink_Eigen;
63
64 // geometry_msgs::PoseStamped goalWRTBaseLink_previous;
65
66 bool firstGetGoal = true;
67
68 // variable to recover offset angle
69 bool FIRST = true;
70 float Delta_offset;
71 Eigen::Vector3d j(0, 1, 0);
72 float Phi_dot_offset;
73 float count_offset_angle = 0;
```

```
74 float num_commands, step_angle_recovered;
75 //------------------------
76
77 Eigen::Vector3d direction_motion, dir_dx, dir_sx;
78
79 Eigen::Vector3d setVel, displacementVec;
80 float sinAlpha, cosAlpha, MagnDisplacement,
     Magn_nextGoalWRTBaseLink; // Magn = Magnitude of a
     vector
81 float start_decrease_velocity = 2.,
     start_decrease_velocity_obstacle = 1., min_vel =
     0.1, max_vel = 0.32;
82 float scaleVelFactor;
83 float DeltaTime, DeltaPhi_vel, frequency = 100;
84 double distance2goal; // It is published and subscribed
     by lidar node to set the variable distance_looked
     by the laser (look script lidar.cpp)
85
86 //ros::ServiceClient distance_client;
87 //madar::SetDistance distanceMsg;
88
89 ros::Publisher pubVelocity;
90 geometry_msgs::Vector3Stamped avoidanceVel;
91
92 //FUNCTIONS
93 //
     ------------------------------------------------------------
     FUNCTION TO CONVERT VECTORS
94 void msgsToEigen(const geometry_msgs::PoseStamped &msgs
     , Eigen::Vector3d &vect){
95   vect(0) = msgs.pose.position.x;
96   vect(1) = msgs.pose.position.y;
97   vect(2) = msgs.pose.position.z;
98 }
99
100 void floatToEigen(const float *vect, Eigen::Vector3d &
     Evect){
101   Evect(0) = vect[0];
102   Evect(1) = vect[1];
103   Evect(2) = vect[2];
104 }
```

```
105
106 //
    FUNCTION TO ROTATE VECTOR
107 void computeRotationMatrix(const double angle,
108                            const Eigen::Vector3d &
                               directionVector,
109                            const Eigen::Vector3d &
                               rotationPoint,     // 0 0
                               0
110                            Eigen::Matrix4d &
                               rotationMatrix){
111
112     double L = directionVector.norm();
113
114     double u = directionVector(0)/L;
115     double v = directionVector(1)/L;
116     double w = directionVector(2)/L;
117
118     double a = rotationPoint(0);
119     double b = rotationPoint(1);
120     double c = rotationPoint(2);
121
122     double u2 = u * u;
123     double v2 = v * v;
124     double w2 = w * w;
125     double cosT = cos(angle);
126     double oneMinusCosT = 1-cosT;
127     double sinT = sin(angle);
128
129     rotationMatrix(0,0) = u2 + (v2 + w2) * cosT;
130     rotationMatrix(0,1) = u * v * oneMinusCosT - w *
          sinT;
131     rotationMatrix(0,2) = u * w * oneMinusCosT + v *
          sinT;
132     rotationMatrix(0,3) = (a*(v2 + w2) - u*(b*v + c*w))
          *oneMinusCosT
133           + (b*w - c*v)*sinT;
134
135     rotationMatrix(1,0) = u * v * oneMinusCosT + w *
          sinT;
```

```
136     rotationMatrix(1,1) = v2 + (u2 + w2) * cosT;
137     rotationMatrix(1,2) = v * w * oneMinusCosT - u *
           sinT;
138     rotationMatrix(1,3) = (b*(u2 + w2) - v*(a*u + c*w))
           *oneMinusCosT
139             + (c*u - a*w)*sinT;
140
141     rotationMatrix(2,0) = u * w * oneMinusCosT - v *
           sinT;
142     rotationMatrix(2,1) = v * w * oneMinusCosT + u *
           sinT;
143     rotationMatrix(2,2) = w2 + (u2 + v2) * cosT;
144     rotationMatrix(2,3) = (c*(u2 + v2) - w*(a*u + b*v))
           *oneMinusCosT
145             + (a*v - b*u)*sinT;
146
147     rotationMatrix(3,0) = 0.0;
148     rotationMatrix(3,1) = 0.0;
149     rotationMatrix(3,2) = 0.0;
150     rotationMatrix(3,3) = 1.0;
151 }
152
153 void computeRotatedPoint(const Eigen::Matrix4d &
       rotationMatrix,
154                          const Eigen::Vector3d &point,
155                          Eigen::Vector3d &rotatedPoint)
                               {
156     Eigen::Vector4d ePoint;
157     ePoint(0)=point(0);
158     ePoint(1)=point(1);
159     ePoint(2)=point(2);
160     ePoint(3)=1.;
161     Eigen::Vector4d rPoint = rotationMatrix * ePoint;
162     rotatedPoint = rPoint.head<3>();
163 }
164
165 //
      FUNCTION TO CHANGE RAFERANCE FRAME of the vector
      GOAL
166 //
```

```
    and BUILD the vectors for THE SPACE COVER by the
    robot
167 //

    during a path which go directly to the goal
168 /*
169         Change the goal vector from world reference frame
                to base_link frame
170           --> Inputs: Pose wrt the world frame
171           --> Outputs: Pose wrt the base_link frame
172 */
173
174 void fromWorld2Base_link(geometry_msgs::Pose goalPose){
175
176     // Initializing all the variables required:
177     geometry_msgs::PoseStamped goalWRTWorld;
178
179     goalWRTWorld.pose = goalPose;
180
181     tf2_ros::Buffer tfBuffer;
182     tf2_ros::TransformListener tfListener(tfBuffer);
183
184     // Creating all the TransformStampsubTrasformed
           variables.
185     geometry_msgs::TransformStamped trasf;
186
187
188     // Obtain tf from "base_link" to "world".pose.
           position.x
189     try{
190         trasf = tfBuffer.lookupTransform( "base_link",
                "world", ros::Time(0), ros::Duration(3.0));
191     }catch (tf2::TransformException &ex) {
192         ROS_WARN("%s", ex.what());
193     }
194
195     // Changing the reference frame of the goal:
196     tf2::doTransform(goalWRTWorld, goalWRTBaseLink,
           trasf);
197
```

```
198         //   if(firstGetGoal){
199         //        goalWRTBaseLink_previous.pose.position.x =
                goalWRTBaseLink.pose.position.x;
200         //        goalWRTBaseLink_previous.pose.position.y =
                goalWRTBaseLink.pose.position.y;
201         //        goalWRTBaseLink_previous.pose.position.z =
                goalWRTBaseLink.pose.position.z;
202
203         //        goalWRTBaseLink_previous.pose.orientation.x
                 = goalWRTBaseLink.pose.orientation.x;
204         //        goalWRTBaseLink_previous.pose.orientation.y
                 = goalWRTBaseLink.pose.orientation.y;
205         //        goalWRTBaseLink_previous.pose.orientation.z
                 = goalWRTBaseLink.pose.orientation.z;
206         //        goalWRTBaseLink_previous.pose.orientation.w
                 = goalWRTBaseLink.pose.orientation.w;
207
208         //        firstGetGoal = false;
209         //   }
210
211
212         msgsToEigen(goalWRTBaseLink, goalWRTBaseLink_Eigen)
                ;
213
214
215         //    ROS_INFO_STREAM("Reference to thsubTrasforme
                base_link:");
216         //    ROS_INFO_STREAM("X: " << goalWRTBaseLink.pose
                .position.x);
217         //    ROS_INFO_STREAM("Y: " << goalWRTBaseLink.pose
                .position.y);
218         //    ROS_INFO_STREAM("Z: " << goalWRTBaseLink.pose
                .position.z);
219         //    ROS_INFO_STREAM("Q.X: " << goalWRTBaseLink.
                pose.orientation.x);
220         //    ROS_INFO_STREAM("Q.Y: " << goalWRTBaseLink.
                pose.orientation.y);
221         //    ROS_INFO_STREAM("Q.Z: " << goalWRTBaseLink.
                pose.orientation.z);
222         //    ROS_INFO_STREAM("Q.W: " << goalWRTBaseLink.
                pose.orientation.w);
```

```
223
224 }
225
226 //
      ------------------------------------------------------------------------------
        MAIN FUNCTION
227 bool X1 = true;
228
229 //--------------------
230 Eigen::Vector3d vec_min;
231 Eigen::Matrix4d rotMatrix_vec_min_dx,
      rotMatrix_vec_min_sx;
232 Eigen::Vector3d point(0,0,0), k(0, 0, 1);
233
234 //--------------------
235
236 void get_boundaries(const madar::vectorArray &msg) {
237
238     bool direction_found = false;
239
240     bool stop_generic_distance = false; // one of the
        laser measure is under the security
241     bool stop_goal_reached = false; // we have reached
        the goal
242     bool stop_boundary_gap = false; // we stop beacause
         one of the boundaries of the gap is so much
        near the robot
243     float stop_dist = 0.415;
244     float stop_goal = 0.1;
245
246     float auxiliar_ang, auxiliar_dist, current_angle;
247     float x_platform, y_platform;
248
249     // declarations
250     float door_vec[2], d_mp[3], d_mp2g[2], door_width;
251     float smallest_path_length = 100, path_length,
        index_best_gap;
252     float min_dist_measure = 1000;
253     Eigen::Vector3d R, L;
254
255     float angleDx, angleSx;
```

```
256
257     gapList.resize(0);
258     laserList.resize(0);
259     std::vector<float> alpha;
260
261     // input goal pose
262     goalPoseWorld.position.x = 0.;
263     goalPoseWorld.position.y = 3.;
264     goalPoseWorld.position.z = 0.0;
265     goalPoseWorld.orientation.x = 0.0;
266     goalPoseWorld.orientation.y = 0.0;
267     goalPoseWorld.orientation.z = 0.0;
268     goalPoseWorld.orientation.w = 1.0;
269     // all the time I can obtain the goal vector wrt
          the base_link!
270     fromWorld2Base_link(goalPoseWorld);
271
272     float goal[2];
273     goal[0] = goalWRTBaseLink_Eigen(0);
274     goal[1] = goalWRTBaseLink_Eigen(1);
275     alphaGoal = atan2(goal[1], goal[0]);
276
277     if (alphaGoal < 0)
278         alphaGoal = alphaGoal + 360 * deg2rad;
279
280     alpha1 = alphaGoal - 90 * deg2rad;
281     alpha4 = alphaGoal + 90 * deg2rad;
282     alpha2 = alphaGoal - atan2(d_safe, sqrt(pow(goal
          [0], 2.) + pow(goal[1], 2.)));
283     alpha3 = alphaGoal + atan2(d_safe, sqrt(pow(goal
          [0], 2.) + pow(goal[1], 2.)));
284
285     std::cout << "1) catch the boundaries " << std::
          endl;
286     // 1) catch the boundaries
287     // Input: msg
288     // Output: laserList, gapList, vec_min
289     bool left_jump_caught=false, right_jump_caught=
          false;
290
291 //     std::cout << "msg size: " << msg.array.size() <<
```

```
    std::endl;
292   for(int i = 1; i < msg.array.size(); i++){
293       previous_magnitude = sqrt(pow(msg.array[i-1].x
              ,2.)+ pow(msg.array[i-1].y,2.));
294       current_magnitude = sqrt(pow(msg.array[i].x,2.)
              + pow(msg.array[i].y,2.));
295       jump = current_magnitude - previous_magnitude;
296
297       //---- store in laserList the range of angle to
              COMPARE with 'space covered by the platform
              '
298
299       // Change of the coordinates: distances
             measured from frame "laser" to frame "
             base_link"
300       x_platform = msg.array[i].y;
301       y_platform = radius_platform + msg.array[i].x;
302       current_angle = atan2(y_platform, x_platform);
303
304       if(current_angle < 0)
305           current_angle = current_angle + 360 *
                deg2rad;
306
307
308       // WARNING!!!
309       // LASERLIST FILTERING in the circle of radius
             = d_safe around the platform:
310       // 'I assume that the platform will be always
             in a free space that is a circle of radius
             d_safe'
311       // the filtering is in the following IF in: &&
             sqrt(pow(x_platform,2.) + pow(y_platform,2.)
             ) >= (radius_platform + 0.02)
312
313       bool no_filtered = false;
314
315       if(alpha1 <= current_angle && current_angle <=
             alpha4 && sqrt(pow(x_platform,2.) + pow(
             y_platform,2.)) >= (radius_platform + 0.04))
             {
316           my_laser.angle = current_angle;
```

```
317             my_laser.distance = sqrt(pow(x_platform,2.)
                    + pow(y_platform,2.));
318             laserList.push_back(my_laser);
319             no_filtered = true;
320         }
321         //----
322
323         if (no_filtered){
324
325             if(my_laser.distance <= min_dist_measure){
326                 vec_min(0) = x_platform;
327                 vec_min(1) = y_platform;
328                 vec_min(2) = 0;
329                 min_dist_measure = vec_min.norm();
330             }
331
332             if(left_jump_caught && right_jump_caught){
333                 left_jump_caught = false;
334                 right_jump_caught = false;
335                 gapList.push_back(my_gap);
336             }
337
338             if(jump < - platform_width && !
                   left_jump_caught && right_jump_caught){
339                 left_jump_caught = true;
340                 // these vectors are expressed wrt
                       base_link (not wrt laser frame)
341                 my_gap.left_boundary_vec[0] = msg.array
                       [i].y;
342                 my_gap.left_boundary_vec[1] =
                       radius_platform + msg.array[i].x;
343             }
344
345             else if(jump > platform_width && !
                   right_jump_caught){
346                 right_jump_caught = true;
347                 // these vectors are expressed wrt
                       base_link (not wrt laser frame)
348                 my_gap.right_boundary_vec[0] = msg.
                       array[i-1].y;
349                 my_gap.right_boundary_vec[1] =
```

```
                    radius_platform + msg.array[i-1].x;
350             }
351         }
352     }
353
354     // Sort laserList 'ascending order' (booble sort)
355
356 //     std::cout << "laseList size: " << laserList.size
        () << std::endl;
357     for (int i = 0; i < laserList.size()-1; i++){
358         for(int j = 0; j < laserList.size()-i-1; j++) {
359             if (laserList[j].angle > laserList[j+1].
                angle){
360                 auxiliar_ang = laserList[j].angle;
361                 auxiliar_dist = laserList[j].distance;
362
363                 laserList[j].angle = laserList[j+1].
                    angle;
364                 laserList[j].distance = laserList[j+1].
                    distance;
365
366                 laserList[j+1].angle = auxiliar_ang;
367                 laserList[j+1].distance = auxiliar_dist
                    ;
368             }
369         }
370     }
371     // Save laserList in file
372     if (X1){
373         //                  X1 = false;
374         std::ofstream myfile ("laserList.txt");
375         if (myfile.is_open()){
376             std::cout<<"FILE 'laserList.txt' OPENED" <<
                std::endl;
377
378             for (int i = 0; i < laserList.size(); i++){
379                 myfile << 1/deg2rad * laserList[i].
                    angle << " " << laserList[i].
                    distance <<std::endl;
380             }
381
```

```
382             myfile.close();
383         }
384         else std::cout << "Unable to open file";
385     }
386
387
388
389
390
391     std::cout << "2) Parametrization" << std::endl;
392     // 2) Parametrization
393     // Input: laserList
394     // Output: alpha, d_amm
395
396
397     float d_amm[laserList.size()];
398     float my_alpha = laserList[0].angle;
399     alpha.push_back(my_alpha);
400     d_amm[0] = d_safe/cos(alpha[0] - alpha1);
401     for(int i = 0; i < laserList.size()-1; i++){
402         my_alpha += (laserList[i+1].angle - laserList[i
              ].angle);
403         alpha.push_back(my_alpha);
404
405         if(alpha1 <= alpha[i] && alpha[i] < alpha2)
406             d_amm[i] = d_safe/cos(alpha[i] - alpha1);
407
408         else if(alpha2 <= alpha[i] && alpha[i] <
              alphaGoal){
409             theta = asin((sqrt((pow(goal[0], 2.) + pow(
                  goal[1], 2.))) * sin(alpha1 + 90*deg2rad
                   - alpha[i]))
410                     /d_safe);
411             d_amm[i] = d_safe * cos(theta) + sqrt((pow(
                  goal[0], 2.) + pow(goal[1], 2.))) * cos(
                  alpha1 + 90*deg2rad - alpha[i]);
412         }
413         else if (alphaGoal <= alpha[i] && alpha[i] <
              alpha3){
414             theta = asin((sqrt((pow(goal[0], 2.) + pow(
                  goal[1], 2.))) * sin(alpha[i] - 90*
```

```
                deg2rad - alpha1))
415                  /d_safe);
416             d_amm[i] = d_safe * cos(theta) + sqrt((pow(
                    goal[0], 2.) + pow(goal[1], 2.))) * cos(
                    alpha[i] - 90*deg2rad - alpha1);
417         }
418         else if (alpha3 <= alpha[i] && alpha[i] <
                alpha4)
419             d_amm[i] = d_safe/cos(alpha4 - alpha[i]);
420     }
421     alpha.push_back (laserList[laserList.size()-1].
            angle);
422     d_amm[laserList.size()-1] = d_safe/cos(alpha4 -
            alpha[laserList.size()-1]);
423     // Save parametrization in file
424     if (W){
425         //                  W = false;
426         std::ofstream myfile ("distance_ammisibile.txt"
                );
427         if (myfile.is_open()){
428             std::cout<<"FILE 'distance_ammisibile.txt'
                    OPENED" << std::endl;
429
430             for(int count = 0; count < laserList.size()
                    ; count ++){
431                 myfile <<  1/deg2rad * alpha[count] <<
                        " " << d_amm[count] << " " << std::
                        endl;
432             }
433
434             myfile.close();
435         }
436         else std::cout << "Unable to open file";
437     }
438
439
440     std::cout << "3) Choose the best door" << std::endl
            ;
441     // 3) Choose the best door
442     // Input: gapList, goalWRTBaseLink
443     // Output: index_BestGap
```

```
444     float path_length1, path_length2, path_length3;
445     for(int j = 0; j < gapList.size(); j++){
446         door_width = sqrt(pow(gapList[j].
                left_boundary_vec[0] - gapList[j].
                right_boundary_vec[0], 2.) +
447                 pow(gapList[j].left_boundary_vec[1] -
                        gapList[j].right_boundary_vec[1],
                        2.));
448
449         if (door_width > platform_width){
450
451             door_vec[0] = gapList[j].left_boundary_vec
                    [0] - gapList[j].right_boundary_vec[0];
452             door_vec[1] = gapList[j].left_boundary_vec
                    [1] - gapList[j].right_boundary_vec[1];
453
454             // LOGIC 1 to choose the best door: using
                    the middle point of the door
455             d_mp[0] = gapList[j].right_boundary_vec[0]
                    + door_vec[0]/2;
456             d_mp[1] = gapList[j].right_boundary_vec[1]
                    + door_vec[1]/2;
457             d_mp[2] = 0;
458
459             d_mp2g[0] = goalWRTBaseLink.pose.position.x
                    - d_mp[0];
460             d_mp2g[1] = goalWRTBaseLink.pose.position.y
                    - d_mp[1];
461
462             path_length3 = sqrt(pow(d_mp[0], 2.) + pow(
                    d_mp[1], 2.)) + sqrt(pow(d_mp2g[0], 2.)
                    + pow(d_mp2g[1], 2.));
463
464             // LOGIC 2 to choose the best door: using
                    the boundary points of the door
465             // new logic to choose the door
466             // path = min (path1, path2)
467             // path1 = |vec(R)| + |vec(G) + vec(R)|
468             // path2 = |vec(L)| + |vec(G) + vec(L)|
469             path_length1 = sqrt(pow(gapList[j].
                    right_boundary_vec[0],2.) + pow(gapList[
```

```
            j].right_boundary_vec[1],2.)) +
470                 sqrt(pow((goalWRTBaseLink_Eigen(0)
                        - gapList[j].right_boundary_vec
                        [0]), 2.) +
471                 pow((goalWRTBaseLink_Eigen(1) -
                        gapList[j].right_boundary_vec
                        [1]),2.));
472         path_length2 = sqrt(pow(gapList[j].
            left_boundary_vec[0],2.) + pow(gapList[j
            ].left_boundary_vec[1],2.)) +
473                 sqrt(pow((goalWRTBaseLink_Eigen(0)
                        - gapList[j].left_boundary_vec
                        [0]), 2.) +
474                 pow((goalWRTBaseLink_Eigen(1) -
                        gapList[j].left_boundary_vec[1])
                        ,2.));
475
476         if (path_length1 <= path_length2){
477             path_length = path_length1;
478             if(path_length3 <= path_length1)
479                 path_length = path_length3;
480         }
481         else{
482             path_length = path_length2;
483             if(path_length3 <= path_length2)
484                 path_length = path_length3;
485
486         }
487         if (path_length < smallest_path_length){
488             smallest_path_length = path_length;
489             index_best_gap = j;
490             d_safe_vec[0] = d_safe * door_vec[0] /
                    (sqrt(pow(door_vec[0], 2.) + pow(
                    door_vec[1], 2.)));
491             d_safe_vec[1] = d_safe * door_vec[1] /
                    (sqrt(pow(door_vec[0], 2.) + pow(
                    door_vec[1], 2.)));
492             //                  d_mp_best_gap(0) =
                    d_mp[0];
493             //                  d_mp_best_gap(1) =
                    d_mp[1];
```

```
                    //                    d_mp_best_gap(2) = 0;
                }
            }
        }

        // !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
        if(vec_min.norm() <= stop_dist){
            stop_generic_distance = true;
            std::cout << "STOP: generic laser's distance so much small" << std::endl;
        }

        // if the smallest distance is: stop_dist < vec_min.norm() <= d_safe  ---> move orthogonal to vec_min in right

        // CASE 0: orthogonal to vec_min

        std::cout << stop_dist << " < " << vec_min.norm() << " <= " << d_safe << std::endl;
        //     Eigen::Vector3d crossDx,crossSx;


        if(stop_generic_distance){
            // STOP

            std::cout << "stop_generic_distance = " << stop_generic_distance << std::endl;

            // -->> Do nothing
        }else{
            // CASES

            // CASE 0
            if(vec_min.norm() <= d_safe){
                std::cout << stop_dist << " < " << vec_min.norm() << " <= " << d_safe << std::endl;
                std::cout << "stop_dist < vec_min NORM <= d_safe" << std::endl;
```

```
526             std::cout << std::endl << "CASE 0:
                    Orthogonal to VEC_MIN" << std::endl;
527
528             computeRotationMatrix(-M_PI_2, k, point,
                    rotMatrix_vec_min_dx);
529             computeRotatedPoint(rotMatrix_vec_min_dx,
                    vec_min, dir_dx);
530             dir_dx = dir_dx.normalized();
531             angleDx = acos(goalWRTBaseLink_Eigen.
                    normalized().dot(dir_dx));
532
533             computeRotationMatrix(+M_PI_2, k, point,
                    rotMatrix_vec_min_sx);
534             computeRotatedPoint(rotMatrix_vec_min_sx,
                    vec_min, dir_sx);
535             dir_sx = dir_sx.normalized();
536             angleSx = acos(goalWRTBaseLink_Eigen.
                    normalized().dot(dir_sx));
537
538 //              std::cout << "angleDx = " << angleDx <<
    std::endl;
539 //              std::cout << "angleSx = " << angleSx <<
    std::endl;
540
541             if(angleDx <= angleSx){
542                 std::cout << "orthogonal DX" << std::
                        endl;
543                 direction_motion(0) = dir_dx(0);
544                 direction_motion(1) = dir_dx(1);
545             }
546             else{
547                 std::cout << "orthogonal SX" << std::
                        endl;
548                 direction_motion(0) = dir_sx(0);
549                 direction_motion(1) = dir_sx(1);
550             }
551             direction_found = true;
552         }else if (gapList.size() != 0){
553
554             R(0) = gapList[index_best_gap].
                    right_boundary_vec[0];
```

```
555             R(1) = gapList[index_best_gap].
                    right_boundary_vec[1];
556             R(2) = 0;
557
558             L(0) = gapList[index_best_gap].
                    left_boundary_vec[0];
559             L(1) = gapList[index_best_gap].
                    left_boundary_vec[1];
560             L(2) = 0;
561
562             right_boundary_vec_LIM[0] = gapList[
                    index_best_gap].right_boundary_vec[0] +
                    d_safe_vec[0];
563             right_boundary_vec_LIM[1] = gapList[
                    index_best_gap].right_boundary_vec[1] +
                    d_safe_vec[1];
564             right_boundary_vec_LIM[2] = 0;
565
566             left_boundary_vec_LIM[0] = gapList[
                    index_best_gap].left_boundary_vec[0] -
                    d_safe_vec[0];
567             left_boundary_vec_LIM[1] = gapList[
                    index_best_gap].left_boundary_vec[1] -
                    d_safe_vec[1];
568             left_boundary_vec_LIM[2] = 0;
569
570
571             // 2 bis) txt producermsgsToEigen
572             if (X){
573                 //                  X = false;
574                 std::ofstream myfile ("
                        vec_lim_and_goal_position.txt");
575                 if (myfile.is_open()){
576                     std::cout<<"FILE '
                            vec_lim_and_goal_position.txt'
                            OPENED" << std::endl;
577
578                     myfile << sqrt(pow(R(0), 2.) + pow(
                            R(1), 2.)) << " "
579                         << 1/deg2rad * atan2 (R(1) ,
                                R(0)) << " "
```

```
580                         << sqrt(pow(L(0), 2.) + pow(
                               L(1), 2.)) << " "
581                         << 1/deg2rad * atan2 (L(1) ,
                                L(0)) << " "
582                         << sqrt(pow(goalWRTBaseLink.
                               pose.position.x, 2.) +
                               pow(goalWRTBaseLink.pose.
                               position.y, 2.)) << " "
583                         << 1/deg2rad * atan2 (
                               goalWRTBaseLink.pose.
                               position.y,
                               goalWRTBaseLink.pose.
                               position.x) << " " <<std
                               ::endl;
584
585                     myfile.close();
586                 }
587                 else std::cout << "Unable to open file"
                        ;
588             }
589
590
591             // print gapList size and index of the gap
                    which provides the smallest path
592             //    std::cout<<"gapList size " << gapList
                    .smsgsToEigenize() << std::endl;
593             //    std::cout<<"index best gap " <<
                    index_best_gap << std::endl;
594
595
596             if (first_execution_front){
597                 //                  first_execution_front
                        =false;
598                 std::ofstream myfile ("gap.txt");
599                 if (myfile.is_open()){
600                     std::cout << "FILE 'gap.txt' OPENED
                            " << std::endl;
601                     for(int count = 0; count < gapList.
                            size(); count ++){
602
603                         myfile <<  sqrt(pow(gapList[
```

```
                                    count].left_boundary_vec[0],
                                     2.) + pow(gapList[count].
                                    left_boundary_vec[1], 2.))
                                    << " "

                                    << 1/deg2rad * atan2 (
                                    gapList[count].
                                    left_boundary_vec[1] ,
                                    gapList[count].
                                    left_boundary_vec[0]) << " "


                                    << sqrt(pow(gapList[count].
                                    right_boundary_vec[0], 2.) +
                                     pow(gapList[count].
                                    right_boundary_vec[1], 2.))
                                    << " "

                                    << 1/deg2rad * atan2(gapList
                                    [count].right_boundary_vec
                                    [1], gapList[count].
                                    right_boundary_vec[0]) << "
                                    "

                                    << std::endl;
604                         }
605                         myfile.close();
606                     }
607                     else std::cout << "Unable to open file"
                            ;
608                 }
609 //-------------------------------------------------------------
610                 // Compute all the vectors you need to
                       choose the best direction
611                 // that is choose the CASE (1 - 4,4bis) in
                       which the robot is.
612
613                 // variables declaration
614                 Eigen::Vector3d goalWRTBaseLink_Eigen,
                       left_boundary_vec_LIM_Eigen, crossVect,
615                         right_boundary_vec_LIM_Eigen, dir1,
```

```
                                dir2;
616             Eigen::Vector3d R_tan, L_tan;
617
618             // convertions
619             msgsToEigen(goalWRTBaseLink,
                    goalWRTBaseLink_Eigen);
620             floatToEigen(left_boundary_vec_LIM,
                    left_boundary_vec_LIM_Eigen);
621             floatToEigen(right_boundary_vec_LIM,
                    right_boundary_vec_LIM_Eigen);
622
623             // TANGENT VECTORs: compute vectors tangent
                     to a virtual circonference centered in
                    Right and Left boundary of a vector
624             Eigen::Matrix4d rotMatrix_R, rotMatrix_L;
625             float magn_R_tan, magn_L_tan, rot_angle_R,
                    rot_angle_L;
626
627             magn_R_tan = sqrt(pow(R.norm(), 2.) - pow(
                    d_safe, 2.));
628             magn_L_tan = sqrt(pow(L.norm(), 2.) - pow(
                    d_safe, 2.));
629
630             //     rot_angle_R = acos(magn_R_tan/R.norm
                    ());
631             rot_angle_R = atan2(d_safe, R_tan.norm());
632 //              std::cout << "rot_angle_R = " <<
    rot_angle_R << std::endl;
633
634             //     rot_angle_L = acos(magn_L_tan/L.norm
                    ());
635             rot_angle_L = atan2(d_safe, L_tan.norm());
636 //              std::cout << "rot_angle_L = " <<
    rot_angle_L << std::endl;
637
638             // the variable point is declared before
                    and it is: point(0, 0, 0)
639             computeRotationMatrix(rot_angle_R, -k,
                    point, rotMatrix_R);
640             computeRotatedPoint(rotMatrix_R, R, R_tan);
641             R_tan = R_tan.normalized() * magn_R_tan;
```

```
642
643             computeRotationMatrix(rot_angle_L, k, point
                    , rotMatrix_L);
644             computeRotatedPoint(rotMatrix_L, L, L_tan);
645             L_tan = L_tan.normalized() * magn_L_tan;
646 //
    ------------------------------------------------------------------------

647             // For all cases
648             direction_motion(2) = 0; // direction
                   motion in z-axis direction is null
649
650             // Check if platform can go directly to the
                    goal
651             bool directionToGoal = true;
652
653             for(int i = 0; i < laserList.size(); i++){
654                 if(laserList[i].distance < d_amm[i]){
655                     directionToGoal = false;
656                     break;
657                 }
658             }
659
660             // CASE 1: Directly to the goal
661             if(directionToGoal){
662                 std::cout << std::endl << "CASE 1 to
                        the goal" << std::endl;
663                 direction_motion =
                        goalWRTBaseLink_Eigen.normalized();
664             }
665             else{
666                 // CASE 2: GOAL on the LEFT of the DOOR
667                 if((k.dot(goalWRTBaseLink_Eigen.cross(
                        L_tan))) <= 0
668                         && (k.dot(goalWRTBaseLink_Eigen
                            .cross(R_tan))) <= 0){
669
670                     std::cout << std::endl << "CASE 2 =
                            goal on the LEFT";
671
672                     if (k.dot(L_tan.cross(R_tan)) >= 0)
```

```
                {
673                 direction_motion = R_tan.
                        normalized();
674                 std::cout<<" R tangent " << std
                        ::endl;
675             }
676             else{
677                 if(L_tan.norm() >= R_tan.norm()
                        ){
678                     direction_motion = R_tan.
                            normalized();
679                     std::cout<<" R tangent " <<
                            std::endl;
680                 }
681                 else{
682                     direction_motion = L_tan.
                            normalized();
683                     std::cout<<" L tangent " <<
                            std::endl;
684                 }
685             }
686
687         }
688         // CASE 3: GOAL on the RIGHT of the
               DOOR
689         else if((k.dot(goalWRTBaseLink_Eigen.
                cross(L_tan))) >= 0
690                 && (k.dot(goalWRTBaseLink_Eigen
                        .cross(R_tan))) >= 0){
691
692             std::cout << std::endl << "CASE 3 =
                    goal on the RIGHT";
693
694             if (k.dot(L_tan.cross(R_tan)) > 0){
695                 std::cout<<" L tangent" << std
                        ::endl;
696                 direction_motion = L_tan.
                        normalized();
697             }
698             else{
699                 if(R_tan.norm() >= L_tan.norm()
```

```
                    ){
700                     direction_motion = L_tan.
                            normalized();
701                     std::cout<<" L tangent " <<
                            std::endl;
702                 }
703                 else{
704                     direction_motion = R_tan.
                            normalized();
705                     std::cout<<" R tangent " <<
                            std::endl;
706                 }
707             }
708         }
709         // CASE 4: GOAL in the MIDDLE
710         else if((k.dot(goalWRTBaseLink_Eigen.
                cross(L_tan))) >= 0
711                 && (k.dot(goalWRTBaseLink_Eigen
                        .cross(R_tan))) <= 0){
712
713             std::cout << std::endl << "CASE 4 =
                    goal in the middle";
714
715             if(L_tan.norm() >= R_tan.norm()){
716                 direction_motion = R_tan.
                        normalized();
717                 std::cout<<" R tangent " << std
                        ::endl;
718             }
719             else{
720                 direction_motion = L_tan.
                        normalized();
721                 std::cout<<" L tangent " << std
                        ::endl;
722             }
723
724         }
725         // CASE 4bis: GOAL in the MIDDLE
726         else if((k.dot(goalWRTBaseLink_Eigen.
                cross(L_tan))) <= 0
727                 && (k.dot(goalWRTBaseLink_Eigen
```

```
                                .cross(R_tan))) >= 0){
728
729                         std::cout << std::endl << "CASE 4
                               bis = goal in the middle, bis"
                               << std::endl;
730
731                         direction_motion(0) =
                               goalWRTBaseLink_Eigen(0);
732                         direction_motion(1) =
                               goalWRTBaseLink_Eigen(1);
733                         direction_motion(2) = 0;
734                         direction_motion = direction_motion
                               .normalized();
735
736 //                          if(L_tan.norm() >= R_tan.norm()){
737 //                              direction_motion = R_tan.
    normalized();
738 //                              std::cout<<" R tangent " <<
    std::endl;
739 //                          }
740 //                          else{
741 //                              direction_motion = L_tan.
    normalized();
742 //                              std::cout<<" L tangent " <<
    std::endl;
743 //                          }
744
745                     }
746                     // CASE 5: inside d_safe distance of a
                           boundary vector
747 //                    else{
748 //                        Eigen::Matrix4d rotMatrix_case5;
749
750 //                        std::cout << "R norm = " << R.
    norm() << std::endl;
751 //                        std:: cout << "L norm = " << L.
    norm() << std:: endl;
752 //                        if (stop_dist <= R.norm()  && R.
    norm() <= d_safe){
753 //                            std::cout << "CASE 4bisbis:
    move orthogonal, RIGHT" << std::endl;
```

```
754 //                          computeRotationMatrix(-
    M_PI_2, k, point, rotMatrix_case5);
755 //                          computeRotatedPoint(
    rotMatrix_case5, R, direction_motion);
756 //                          direction_motion =
    direction_motion.normalized();
757 //                      }
758 //                      else if (stop_dist <= L.norm()
    && L.norm() <= d_safe){
759 //                          std::cout << "CASE 4bisbis:
    move orthogonal, LEFT" << std::endl;
760 //                          computeRotationMatrix(M_PI_2,
     k, point, rotMatrix_case5);
761 //                          computeRotatedPoint(
    rotMatrix_case5, L, direction_motion);
762 //                          direction_motion =
    direction_motion.normalized();
763 //                      }
764
765 //                  }
766             }
767         }else{
768             // CASE 5: no gaps detected. The robot will
                    move in the direction of
                goalWRTBaseLink even if it will find
                obstacle during the path
769             // The robo will be able to avoid the
                obstacles which it will find entering in
                 an other CASE during the motion.
770             std::cout << "CASE 5: no gaps detected! I
                decide to move directly to the goal" <<
                std::endl;
771             direction_motion(0) = goalWRTBaseLink_Eigen
                (0);
772             direction_motion(1) = goalWRTBaseLink_Eigen
                (1);
773             direction_motion(2) = 0;
774             direction_motion = direction_motion.
                normalized();
775         }
776
```

```
777         // -->> direction_motion
778
779         // 3bis) txt producer of best direction unit
                vector
780         if (Y){
781             //         Y = false;
782             std::ofstream myfile ("best_direction.txt")
                    ;
783             if (myfile.is_open()){
784                 std::cout<<"FILE 'best_direction.txt'
                        OPENED" << std::endl;
785                 myfile << sqrt(pow(direction_motion(0),
                        2.) + pow(direction_motion(1), 2.))
                        << " "
786                             << 1/deg2rad * atan2(
                                direction_motion(1),
                                direction_motion(0))<<std::
                                endl;
787
788                 myfile.close();
789             }
790             else std::cout << "Unable to open file";
791         }
792     }
793
794     // HERE VELOCITY SETTING
795     if(goalWRTBaseLink_Eigen.norm() <= stop_goal){
796         stop_goal_reached = true;
797         std::cout << "STOP: goal reached" << std::endl;
798     }
799
800     if (stop_goal_reached || stop_boundary_gap ||
            stop_generic_distance){
801         std::cout << "STOP!" << std::endl;
802         setVel(0) = 0;                  // xd    (m/s)
803         setVel(1) = 0;                  // yd    (m/s)
804         setVel(2) = 0;                  // phid (rotation
                rad/s)
805         std::cout << "ERROR in POSITION = " <<
                goalWRTBaseLink_Eigen.norm() << "m" << std::
                endl;
```

```
806             std::cout << "ERROR in ORIENTATION = " << acos(
                    goalWRTBaseLink_Eigen.normalized().dot(j))/
                    deg2rad << "deg" << std::endl;
807         }
808         else {
809             // WARNING!
810             // Be careful to the frame of the robot: it is
                    not the same of the frame in RViz (there is
                    a mistake in the old scripts
811             std::cout << "MOVE!" << std::endl << std::endl;
812 
813             if(goalWRTBaseLink_Eigen.norm() <=
                    start_decrease_velocity){
814                 scaleVelFactor = goalWRTBaseLink_Eigen.norm
                        ()/start_decrease_velocity * max_vel;
815                 if (scaleVelFactor <= min_vel)
816                     scaleVelFactor = min_vel;
817             }
818             else if (vec_min.norm() <=
                    start_decrease_velocity_obstacle){
819                 scaleVelFactor = vec_min.norm()/
                        start_decrease_velocity_obstacle *
                        max_vel;
820                 if (scaleVelFactor <= min_vel)
821                     scaleVelFactor = min_vel;
822             }else
823                 scaleVelFactor = max_vel;
824 
825             std::cout << "scale factor: " << scaleVelFactor
                    << std::endl;
826 
827             setVel(0) = scaleVelFactor * direction_motion
                    (1);                    // xd   (m/s)
828             setVel(1) = - scaleVelFactor * direction_motion
                    (0);                  // yd   (m/s)
829             setVel(2) = 0;
                    // phid (rad/s)
830 
831             // ROTATIONAL VELOCITY (Control of the
                    orientation: always look to the goal)
```

```
832         // WARNING!!! ---> ORIENTATION PROPORTIONAL
                CONTROL
833         // we want to have the vector 'goalWRTBaseLink'
                ALWAYS aligned with Y-axis of the frame '
                base_link'
834         // In order to compute the angle error to
                recover instant for instant we have to
                consider that there are two causes of error
835         // 1) error DeltaTheta dues by INCORRECT
                INITIAL ORIENTATION of the frame 'base_link
                ':
836         //              DeltaTheta_BL
837
838         // 2) error due by the translational velocity
                of the platform set by the algorithm
839         //              DeltaTheta_vel
840
841         // here I compute the rotational velocity in
                order to garantee that the robot looks
                always to the goal during the motion
842
843         // ORIENTATION
844         // 1) ERROR DUE by the TRANSLATIONAL VELOCITY
845         DeltaTime = 1/frequency; // 1:rate
846 //          std::cout << "Delta Time:   " << DeltaTime <<
    std::endl;
847
848         MagnDisplacement = sqrt(pow(setVel(0), 2.) +
                pow(setVel(1), 2.)) * DeltaTime;
849 //          std::cout << "DeltaS magnitude:   " <<
    MagnDisplacement << std::endl;
850
851         displacementVec(0) = direction_motion(0) *
                MagnDisplacement;
852         displacementVec(1) = direction_motion(1) *
                MagnDisplacement;
853         displacementVec(2) = 0;
854
855         cosAlpha = (displacementVec.dot(
                goalWRTBaseLink_Eigen)) / (displacementVec.
                norm() * goalWRTBaseLink_Eigen.norm());
```

```
856 //         std::cout << "cosAlpha:  " << cosAlpha << std
        ::endl;
857
858         sinAlpha = (displacementVec.cross(
                goalWRTBaseLink_Eigen)).norm() / (
                displacementVec.norm() *
                goalWRTBaseLink_Eigen.norm());
859 //         std::cout << "sinAlpha:  " << sinAlpha << std
        ::endl;
860
861         Magn_nextGoalWRTBaseLink = sqrt(pow(
                MagnDisplacement, 2.) + pow(
                goalWRTBaseLink_Eigen.norm(), 2.)
862                                         - 2 *
                                            MagnDisplacement
                                             *
                                            goalWRTBaseLink_Eigen
                                            .norm() *
                                            cosAlpha);
863 //         std::cout << "G':  " << goalWRTBaseLink_Eigen
        .norm() << std::endl;
864 //         std::cout << "G'':  " <<
        Magn_nextGoalWRTBaseLink << std::endl;
865
866         DeltaPhi_vel = asin(MagnDisplacement * sinAlpha
                / Magn_nextGoalWRTBaseLink); // should be
                always > 0 so I need to decide
867
868 //         std::cout << "DeltaPhi_vel:  " <<
        DeltaPhi_vel << std::endl;
869
870
871         float epsilon;
872         epsilon = sin(5*deg2rad); // tollerance
873
874         float Phi_dot_vel;
875
876         // here I decide the sign of the rotational
                velocity
877         if(k.dot((displacementVec.cross(
                goalWRTBaseLink_Eigen)).normalized()) >=
```

```
            epsilon)
878             Phi_dot_vel = DeltaPhi_vel / DeltaTime;
879         else if(k.dot((displacementVec.cross(
            goalWRTBaseLink_Eigen)).normalized()) <= -
            epsilon)
880             Phi_dot_vel = - DeltaPhi_vel / DeltaTime;
881         else if(-epsilon < k.dot((displacementVec.cross
            (goalWRTBaseLink_Eigen)).normalized()) &&
882                 k.dot((displacementVec.cross(
                    goalWRTBaseLink_Eigen)).normalized()
                    ) < epsilon)
883             Phi_dot_vel = 0;
884
885 //          std::cout << "Phi_dot_vel = " << Phi_dot_vel
    << std::endl;
886
887         // 2) ERROR DUE by INCORRECT INITIAL
            ORIENTATION of the frame 'base_link'
888
889         if(FIRST){
890             Delta_offset = acos(goalWRTBaseLink_Eigen.
                normalized().dot(j));
891 //              std::cout << "Delta_offset = " <<
    Delta_offset << std::endl;
892             FIRST = false;
893         }
894         // hp: I supposed to recover the offset in 5
            sec.
895         num_commands = 5/(1/frequency); // #commands in
             5 sec
896 //          std::cout << "num_commands = " <<
    num_commands << std::endl;
897         step_angle_recovered = Delta_offset/
            num_commands; // angle recovered in 1
            command
898 //          std::cout << "step_angle_recovered = " <<
    step_angle_recovered << std::endl;
899
900         if (k.dot(goalWRTBaseLink_Eigen.cross(j)) < 0)
901             Phi_dot_offset = +step_angle_recovered/(1/
                frequency);
```

```
902         else
903             Phi_dot_offset = -step_angle_recovered/(1/
                frequency);
904
905 //          std::cout << "Phi_dot_offset = " <<
    Phi_dot_offset << std::endl;
906
907         if (count_offset_angle < Delta_offset)
908             count_offset_angle += step_angle_recovered;
909 //          std::cout << "count_offset_angle = " <<
    count_offset_angle << std::endl;
910
911         if (Delta_offset > count_offset_angle)
912             setVel(2) =  Phi_dot_vel + Phi_dot_offset;
913         else
914             setVel(2) = Phi_dot_vel;
915
916 //          std::cout << "setVel(2) = " << setVel(2) <<
    std::endl;
917
918      }
919
920 //     std::cout << "x_dot:   " << setVel(0) << std::endl
921 //               << "y_dot: " << setVel(1) << std::endl
922 //               << "Phi_dot: " << setVel(2) << std::
    endl;
923
924      distance2goal = goalWRTBaseLink_Eigen.norm();
925      std::cout << "distance2goal:   " << distance2goal <<
         std::endl;
926
927      //     // CLIENT CALL
928      //     distanceMsg.request.newdistance =
        distance2goal;
929
930      //     if (distance_client.call(distanceMsg)){
931      //         ROS_INFO("Change distance looked");
932      //     }else{
933      //         ROS_ERROR("Failed to call SetDistance");
934      //     }
935
```

```
936
937     avoidanceVel.vector.x = setVel(0);
938     avoidanceVel.vector.y = setVel(1);
939     //     avoidanceVel.vector.z = setVel(2);
940
941
942     // filter for rotational velocity-----
943     float max_rotational_vel = 7*M_PI/180;
944
945     if (setVel(2) > (max_rotational_vel)){
946         if(setVel(2) > 0)
947             avoidanceVel.vector.z = max_rotational_vel;
948         else
949             avoidanceVel.vector.z = -max_rotational_vel
                ;
950     }
951     else
952         avoidanceVel.vector.z = setVel(2);
953     //-----------------------------------------
954
955     std::cout << "x_dot:   " << avoidanceVel.vector.x <<
        std::endl
956               << "y_dot: " << avoidanceVel.vector.y <<
                  std::endl
957               << "Phi_dot: " << avoidanceVel.vector.z
                  << std::endl;
958
959
960     //     avoidanceVel.vector.z = 0;
961
962     pubVelocity.publish(avoidanceVel);
963 }
964 //
    ------------------------------------------------------------------------


965
966 int main(int argc, char **argv)
967 {
968
969     ros::init(argc,argv,"avoidance_algorithm");
970     ros::NodeHandle n;
```

```
971
972     ros::Subscriber subLaserFront = n.subscribe("/
            front_out/scan", 1, get_boundaries);
973
974     // PUBLISHER
975     pubVelocity = n.advertise<geometry_msgs::
            Vector3Stamped>("/command_vel",1);
976
977     //      //CLIENT DECLARATION
978     //      distance_client = n.serviceClient<madar::
            SetDistance>("/setDistance");
979
980
981     //      ros::Rate rate(frequency);
982
983     //      while(ros::ok()) {
984
985
986     //          ros::spinOnce();
987     //          rate.sleep();
988     //      }
989
990     ros::spin();
991 }
```

Bibliography

O. Diegel, A. Badve, G. Bright, J. Potgieter, and S. Tlale. Improved mecanum wheel design for omni-directional robots. In *Proc. 2002 Australasian Conference on Robotics and Automation, Auckland*, pages 117–121, 2002.

S. Y. Ghangrekar. *A path planning and obstacle avoidance algorithm for an autonomous robotic vehicle*. PhD thesis, University of North Carolina at Charlotte, 2009.

R. Philippsen. Motion planning and obstacle avoidance for mobile robots in highly cluttered dynamic environments. Technical report, EPFL, 2004.

J. J. Plumpton, M. J. D. Hayes, R. G. Langlois, and B. V. Burlton. Atlas motion platform mecanum wheel jacobian in the velocity and static force domains. *Transactions of the Canadian Society for Mechanical Engineering*, 38(2):251–261, 2014.

R. Siegwart, I. R. Nourbakhsh, and D. Scaramuzza. *Introduction to autonomous mobile robots*. MIT press, 2011.

R. Suárez, L. Palomo-Avellaneda, J. Martinez, D. Clos, and N. García. Development of a dexterous dual-arm omnidirectional mobile manipulator. *IFAC-PapersOnLine*, 51(22):126–131, 2018.

I. Susnea, V. Minzu, and G. Vasiliu. Simple, real-time obstacle avoidance algorithm for mobile robots. In *8th WSEAS International Conference on Computational Intelligence, Man-Machine Systems and Cybernetics (CIMMACS'09)*, 2009.